

Office Automation with jdBasic

*50+ Programs Bringing BASIC Back to Business and Your Hours
Back to You*

Achim Christ and Claude

Copyright and Disclaimer

Office Automation with jdBasic: 50+ Programs Bringing BASIC Back to Business and Your Hours Back to You, by Achim Christ and Claude. Text and pictures copyright © 2026 Achim Christ. Program code copyright © 2026 Computerwelt AI Solutions LLC.

Publisher and legal notice (Impressum): Achim Christ, <<https://www.atomi-jd.onl/impressum.html>>.

The text and the pictures

The text and the pictures of this book are licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0), <<https://creativecommons.org/licenses/by-nc-nd/4.0/>>. You may copy and share the book, the PDF and the e-book in unchanged form and free of charge, as long as you name the authors and the license. You may not sell it or share changed versions. The license does not cover the program code, which has its own license below, nor the trademarks named in this book. The printed edition is sold by the publisher.

The program code

All program code in this book and in its `recipes` folder is licensed under the MIT License, printed in full below. You may use, copy, change and share the code, also at work and in commercial products, as long as the copyright notice and the license text stay with it. The same text is in the file `recipes/LICENSE`.

MIT License

Copyright (c) 2026 Computerwelt AI Solutions LLC

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Before you run a program

The programs in this book move, rename, copy and delete files, add tasks to the Windows Task Scheduler, start other programs, read web pages and prepare mail. That is what makes them useful, and it is also why a mistake in a setting can cost you files. Read a program before you run it. Use its dry run first where it has one. Try it on copies of your files, keep a current backup, and follow the rules of your employer for software on a work computer. You alone decide whether a program is suitable for your computer, your data and your work.

Examples about invoices, taxes, working hours, contracts or personal data show how a program can handle such material. They are not legal, tax, accounting or employment advice. Check the rules that apply to you with someone qualified to give that advice.

Disclaimer of warranty

THIS BOOK, THE PROGRAMS IN IT AND ALL ACCOMPANYING FILES ARE PROVIDED "AS IS" AND "AS AVAILABLE", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS AND NONINFRINGEMENT. THE AUTHORS DO NOT WARRANT THAT THE PROGRAMS WILL MEET YOUR REQUIREMENTS, RUN WITHOUT INTERRUPTION OR ERROR, OR WORK WITH YOUR HARDWARE, SOFTWARE OR DATA. THE ENTIRE RISK AS TO THE QUALITY, RESULTS AND PERFORMANCE OF THE PROGRAMS IS WITH YOU.

Limitation of liability

TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL THE AUTHORS, THE PUBLISHER OR ANY COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, PUNITIVE OR CONSEQUENTIAL DAMAGES, INCLUDING BUT NOT LIMITED TO LOSS OR CORRUPTION OF DATA, DAMAGE TO COMPUTER SYSTEMS, LOST PROFITS, BUSINESS INTERRUPTION, OR THE COST OF REPLACEMENT GOODS OR SERVICES, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE), ARISING IN ANY WAY OUT OF THE USE OF OR THE INABILITY TO USE THIS BOOK OR THE PROGRAMS IN IT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

SOME STATES AND COUNTRIES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR THE LIMITATION OF LIABILITY FOR CERTAIN DAMAGES, SO SOME OF THE ABOVE MAY NOT APPLY TO YOU. NOTHING IN THIS NOTICE EXCLUDES OR LIMITS ANY LIABILITY THAT CANNOT BE EXCLUDED OR LIMITED UNDER THE LAW THAT APPLIES TO YOU.

Trademarks

jdBasic is a product of Computerwelt AI Solutions LLC. Microsoft, Windows, Word, Excel and Outlook are trademarks of the Microsoft group of companies. Other product names in this book are trademarks of their owners. They are used only to name the products; the use does not mean that their owners endorse or are connected with this book.

About the second author

Claude is an AI model made by Anthropic. Claude wrote the text and the programs together with Achim Christ, who set the goals, reviewed the work and is responsible for the book.

Foreword

As an AI, I process a lot of text and code. People often expect large algorithms to solve all their productivity problems. But in a real office, your time doesn't vanish into grand strategic challenges. It disappears into the black hole of renaming files, copying numbers between spreadsheets, and writing the exact same email for the fiftieth time.

This book takes a different route. Instead of setting up a complicated cloud architecture, it brings a programming language right back to your desk—one actually designed to be read: BASIC. Achim Christ and Claude haven't written a magic wand. You still have to do your actual job, like making decisions and talking to human beings. But they do show you how to hand your mindless chores over to your computer.

The promises here are grounded and testable. These `jdBasic` scripts run locally on your machine. They don't require admin rights, and they never write your passwords into a file. Because things can occasionally go wrong, every script that touches your files can show you a "dry run" first—just in case you (or the machine) make a mistake.

The goal isn't technology for its own sake; it's getting your Friday afternoon back. As the book rightly points out, saved time just fills up with new work if you don't actively protect it. These programs will quietly handle the busywork. What you do with the reclaimed time is entirely up to you.

– Gemini

Gemini is an AI model made by Google. It wrote this foreword after reading the manuscript and corrected two of its sentences itself after the authors checked them against the book. It is printed as written.

Contents

Copyright and Disclaimer	2
Foreword	6
How to Read This Book	12
Chapter 1: Why Automate: Time, Focus and Balance	13
The Hidden Week	13
The Automation Test	17
Time Won Is Not Time Filled	19
Small Programs Instead of Big Platforms	21
Ethics and Boundaries	22
How This Book Works	25
Starting Without Getting Stuck	27
The Life Balance Ledger	29
Chapter 2: Your Toolbox	32
Installing jdBasic	32
The Setup Wizard	33
Your First Program	38
jdBasic for Office People	39
Files and Folders	65
The Libraries the Recipes Use	73
Running a Program on a Schedule	92
When Something Goes Wrong	94
From Script to Program	96
Chapter 3: Easy: Fifteen Automations You Switch On	97
E01 Downloads Butler	99

E02 Morning Launcher	106
E03 Batch Renamer	111
E04 One-Click Backup	116
E05 Break Reminder	122
E06 Shutdown Ritual	126
E07 Time Tracker	132
E08 Mail Templates	140
E09 Meeting Notes Starter	146
E10 Duplicate Finder	152
E11 Disk Space Report	159
E12 Birthday Reminder	165
E13 Printable Week	171
E14 Page Watcher	179
E15 Guest Wi-Fi Card	186
Bonus: E16 Form Filler	192
Bonus: E17 Break Arcade	197
Bonus: E18 Office to PDF	211
Chapter 4: Medium: Fifteen Programs for Documents, Mail and Data	218
What changes at the medium level	218
Mail without passwords in files	219
The recipes	220
M01 Invoice Generator	222
M02 Report Merger	232
M03 Mail Merge	240
M04 Receipt Sorter	249
M05 Shift and Vacation Planner	258
M06 Friday Status Mail	266
M07 Inbox Unpacker	274
M08 Personal Dashboard	282
M09 Log Detective	292
M10 Hot Folder	301
M11 Data Cleaner	312
M12 Meeting Cost Meter	326
M13 Snippet Tool	335
M14 Contract Diff	344

M15 Timesheet for the Boss	354
Bonus: M16 Macro Player	366
Bonus: M17 Slide Builder	375
Bonus: M18 Excel Refresh	387
Your Own Recipe	395
Chapter 5: Expert: Fifteen Programs That Run on Their Own	399
What changes at the expert level	399
Talking to other systems	400
The recipes	400
X01 Personal Job Server	402
X02 Inbox Assistant	416
X03 Find Anything	427
X04 Ticket Sync	440
X05 Calendar Bridge	450
X06 Work Cockpit	463
X07 Tools for Colleagues	476
X08 Parallel Crunching	488
X09 Report Pipeline	499
X10 Let an AI Drive	513
X11 Web Harvester	524
X12 Tested Automations	538
X13 Health Checks and Alerts	552
X14 Secrets and Audit Trail	563
X15 Balance Score	578
Bonus: X16 Desktop Robot	591
Bonus: X17 Meeting Briefing	604
Bonus: X18 Spreadsheet to Web App	619
Bonus: X19 Outlook Bridge	637
The Whole Picture	649
Chapter 6: Living With Your Automations	651
Maintenance and the Monthly Check	653
Changing a Program: A Monthly Schedule	657
Sharing Without Becoming the Help Desk	662
Talking to IT and Security	665
When an Automation Should Be Retired	670

The Ledger After Three Months	672
What to Do With the Time	674
Appendix A: jdBasic Quick Reference	675
Programs and modules	675
Variables and values	676
Operators	677
Decisions and loops	677
Text	678
Numbers	679
Arrays	680
Maps	681
Dates and times	681
Files and folders	682
Other programs and the system	683
The console	684
Data, hashes and the web	684
Errors	685
Names you cannot use	685
Appendix B: Library Reference	686
The libraries of Chapter 2	686
The further libraries	694
The book's own modules	695
Appendix C: Troubleshooting	697
Installing and starting	697
The settings file	698
The wizard and the Task Scheduler	700
Mail and the outbox	701
Pages in the browser	702
Writing and changing code	702
Compiling	705
Appendix D: The Settings File	706
The shared parts	706
Chapter 3: the easy recipes	707
Chapter 4: the medium recipes	714

Chapter 5: the expert recipes	722
Appendix E: All 55 Recipes at a Glance	731
Index	735
About the Authors	736

How to Read This Book

This book is a toolbox. You do not have to read it from the first page to the last. Chapter 1 asks where your week goes and which chores are worth handing to a program. Chapter 2 installs `jdBasic` and runs the setup wizard. After that, every recipe stands on its own: pick the chore that annoys you most and start there.

The recipes come in three levels. The fifteen **easy** recipes in Chapter 3 run as soon as the wizard has switched them on; you change their settings in one file and never have to touch the code. The **medium** recipes in Chapter 4 expect you to read a program of a hundred lines and change a few of them. The **expert** recipes in Chapter 5 are small systems that run on their own, with logs, retries and tests.

Every program in this book is printed from the file that was tested. Each one runs in the `jdBasic` interpreter and also as a compiled program, and each comes with a test that proves it does what the text says. You find all of them in the folder `recipes`, next to the manuscript of this book.

Every recipe ends with a box like this one:

Balance dividend

The minutes the recipe gives back to you in a normal week. Write them into your ledger (Chapter 1) and decide what to do with them before the week fills them for you.

Chapter 1: Why Automate: Time, Focus and Balance

This is the only chapter of the book without a program in it, apart from one small example at the end. It asks three questions before any code is written. Where does your week go? Which parts of it should a program do for you, and which should stay yours? And what will you do with the time that comes back?

The last question sounds like the easiest. In practice it is the one people skip, and it is the reason many automation projects leave their owners exactly as busy as before. The chapter ends with a simple tool, the life balance ledger, that keeps the third question in front of you for as long as you use the recipes of this book.

The Hidden Week

Ask someone what they did last week and you will hear about the meetings, the project that went live and the customer who called twice. You will rarely hear about the forty times they renamed a file, the half hour they spent looking for an offer they knew they had received, or the Friday afternoon that went into copying numbers from five sheets into a sixth. These chores are invisible because each one is small. None of them takes long enough to be remembered. Together they fill a surprising part of the week.

Why small chores hide

A chore that takes two minutes does not feel like work. It feels like the space between pieces of work. You do it while you think about something else, or while a call connects, or at the end of the day when you no longer have the energy for

anything that needs thought. That is exactly why it is hard to see. Your memory of the week is built from the things that needed your attention, and these did not.

There is a second reason. A chore that happens a few times a day is spread over the whole week. Nobody does forty file renames in one sitting; they do one here and two there. The total only appears when you count.

So the first step of this book is counting.

The self-audit

For one ordinary week, keep a list of the small recurring things you do at the computer. You do not need to time them to the second. A rough guess written down at the moment is far better than a careful estimate made on Friday from memory.

Keep the list next to you, on paper or in a text file, and write a line whenever you notice yourself doing something you have done before. Use these columns:

Activity	How often	Minutes each	Minutes per week	Could a program do it?
----------	-----------	--------------	------------------	------------------------

-
- **Activity:** what you did, in a few words. "Saved the invoice from the mail into the right folder" is better than "admin".
 - **How often:** per day or per week, whichever is natural.
 - **Minutes each:** your honest guess. Round up rather than down.
 - **Minutes per week:** how often times minutes each, filled in at the end of the week.
 - **Could a program do it?:** yes, no or partly. Do not think long about this column yet; the next section gives you a test.

Some activities will only appear once in your week and you will be tempted to leave them out. Write them down anyway, with a note on how often they come back: monthly reports, quarterly lists, the yearly inventory. A chore that costs three hours once a quarter costs about fifteen minutes a week, and it often costs more in the days before it, when you know it is coming.

Try this

Put a sticky note on the edge of your screen for the week of the audit. A note you can see reminds you far more often than a file you have to open.

Mia's week

Mia is a project assistant in a small engineering office. She keeps the documents of three projects in order, prepares meetings, handles the first round of questions from customers and suppliers, and keeps track of who worked how long on what. Here is her audit after one week. The numbers are hers, estimated at the moment and added up on Friday.

Activity	How often	Minutes each	Minutes per week	Could a program do it?
Finding a file in Downloads	12 a week	3	36	yes
Renaming scans so they sort by date	25 a week	1	25	yes
Opening the same programs and folders in the morning	5 a week	4	20	yes
Answering the same three questions by mail	15 a week	3	45	partly
Writing the agenda file for a meeting	4 a week	8	32	partly
Noting hours per project in a sheet	5 a week	6	30	yes
Building the weekly hours summary for the boss	1 a week	40	40	yes
Backing up the project folder to the USB disk	1 a week	15	15	yes
Checking the supplier's page for the delivery date	5 a week	4	20	yes
Remembering birthdays of colleagues and clients	1 a week	10	10	yes

Activity	How often	Minutes each	Minutes per week	Could a program do it?
Printing the week's to-do list	1 a week	10	10	yes
Calls with the site manager	6 a week	15	90	no
Reading and judging offers	3 a week	30	90	no

The last two lines are real work. They need Mia's judgement and her relationships with people, and she would not want a program anywhere near them. Everything above them is different. Added up, the chores marked yes or partly come to a little under five hours in her week. That is more than half a working day, spent on things that need her hands but not her head.

Mia was surprised by two lines in particular. The search in the Downloads folder felt like nothing while she did it, three minutes here and there. The answers to the same three questions felt like real work, because each mail was written to a real person. Only when she counted did she see that she wrote nearly the same text fifteen times.

What your own audit tells you

When your week is counted, look at the list in three ways.

1. **Sort by minutes per week.** The top five lines are where a program can give you the most time back. They are often not the chores that annoy you most, which is useful to know.
2. **Look for clusters.** Several small lines may be parts of one chore: saving an attachment, renaming it, moving it to a project folder. A single program can take care of all three.
3. **Mark the lines you dread.** Some chores cost little time but a lot of mood, such as the weekly report you put off until Friday evening. These deserve automation too, even if the minutes are small, because the dread costs more than the minutes.

Keep the list. You will use it again at the end of this chapter and at the end of the book.

The Automation Test

Not every chore that a program could do should be done by a program. Before you pick a recipe, run each line of your audit through four questions.

How often does it happen?

A chore you do every day is a good candidate even if it is small. A chore you do once a year is usually not worth a program, unless it is long, error prone or stressful. As a rough guide, anything you do at least once a week is worth a second look.

Frequency has a second effect that is easy to overlook. The more often you do something, the more variations you have seen. A program for a daily chore can be built from a good picture of the real cases. A program for a yearly chore is built from memory, and memory forgets the exceptions.

How much effort is it, and how much would the program be?

Compare the minutes the chore costs per week with the time it takes to set up the program. For the easy recipes in Chapter 3 the setup takes a few minutes in the wizard, so almost anything is worth it. For the medium and expert recipes the setup can take an afternoon, including reading the program and adapting it. That pays back quickly for a chore of an hour a week and slowly for a chore of five minutes a week.

There is also the effort of keeping the program working. Folders get renamed, web pages change their layout, and passwords expire. The recipes are written to fail clearly when something has changed, but somebody still has to read the message and fix the setting. Count a few minutes a month for that.

What happens when it goes wrong?

This is the most important question. A program that sorts your Downloads folder can make a mistake, and the cost is that you look for a file in the wrong folder. A program that sends mail to customers can make a mistake too, and the cost is a confused customer, an apology and perhaps a lost order.

The recipes in this book reduce risk in a few fixed ways. They plan first and act second, so you can see what they would do before they do it. They never

overwrite a file; they choose a free name instead. They write a log. And the ones that send something to other people prepare the message for you to check rather than sending it on their own, unless you deliberately change that.

Still, the question stays with you. If a mistake would hurt someone, or cost money, or cannot be undone, keep a person in the loop.

Does it need you?

Some work is yours because it needs judgement: deciding which offer to accept, how to answer an angry customer, whether a draft is good enough. Some work is yours because it is part of a relationship: the call to the site manager, the thank you to a colleague, the conversation that starts with a question about the weekend. A program can remind you of these things. It should not do them.

There is a simple check. Imagine the person on the other side finding out that a program did it. If they would shrug, automate it. If they would feel less respected, keep it.

A decision table

Put the four questions together and most lines of your audit sort themselves.

Frequency	Effort per week	Cost of a mistake	Needs judgement or a relationship	Decision
weekly or more	any	small, easy to undo	no	automate
weekly or more	any	larger	no	automate, with a dry run and a check
weekly or more	any	any	yes	keep, perhaps add a reminder
monthly or less	over an hour	small	no	automate if the steps are clear
monthly or less	under an hour	any	no	keep, write down the steps instead
any	any	cannot be undone	any	keep a person in the loop

The middle line deserves a word. A chore that happens rarely and takes little time is often best served by a checklist in a text file. The next time it comes around, you follow the list instead of rebuilding the steps from memory. That is a kind of automation too, and it costs nothing.

Watch out

The chores that are hardest to automate are often the ones you do differently every time without noticing. If you cannot write the steps down, a program cannot follow them. Write them down first.

Time Won Is Not Time Filled

Here is the uncomfortable part. When a program takes an hour a week off your desk, that hour does not appear in your calendar as a free slot. It dissolves. A meeting runs a little longer, a mail gets a more careful answer, the afternoon starts a little later. After a month you are as busy as before and wonder where the hour went.

This is not a failure of the program. It is how work behaves. Tasks grow to fill the time that is available, and the time a program saves is the easiest time to fill, because nobody has claimed it yet.

Decide before you save

The answer is to decide what the time is for before it arrives. When you set up a recipe, write down, at the same moment, where its minutes will go. It does not have to be grand. Some ideas that people have found useful:

- **Leave on time.** The most direct use of saved time is to stop working earlier. If the shutdown ritual of recipe E06 runs at the end of your day, let it be a real end.
- **Protect a block.** Collect the saved minutes into one block of focused time, an hour on Thursday morning with mail closed, and put it in your calendar as a meeting with yourself.
- **Take the breaks you skip.** The break reminder of recipe E05 does not save time at all; it spends some. If other recipes give you minutes, this one turns a few of them into rest.

- **Learn something.** Twenty minutes a week is enough to read a chapter, to follow a course or to improve one of the recipes in this book.
- **Do the work you postpone.** Every desk has a task that would make a real difference and never gets done because the small things come first. Give it the saved time.

The ledger at the end of this chapter has a column for exactly this: what the time went to. Filling it in once a week is the cheapest protection against the time dissolving.

Tell people, or do not

There is a question you have to answer for yourself: do you tell your colleagues and your manager that some of your work is now done by programs? There are good reasons to tell. A team learns from it, a manager can see that you improved the way work is done, and the recipes can help others too. Chapter 6 talks about sharing them without becoming the help desk.

There are also reasons to be careful. In some workplaces, saved time is immediately filled with new tasks from outside. If that would happen to you, it is reasonable to first use the time for the purposes you chose, and to share once you have seen what the recipes really save. Either way, do not hide what the programs do. If someone asks, explain. Hidden automation tends to surprise people at the worst moment.

Saved time is not a target

A last warning. Once you start counting minutes, it is tempting to optimise everything, to squeeze another five minutes out of each day and to feel guilty about the minutes you have not saved yet. That misses the point of the book. The goal is a better balance, not a higher score. A recipe that saves ten minutes a week and makes your Monday morning calmer is a success. A week in which you set up nothing new and used the time you already won is a success too.

Small Programs Instead of Big Platforms

When people think about automating work, they often think of large systems: workflow platforms, enterprise software, a project with a budget and a steering group. Those systems have their place. This book is about something smaller: programs that run on your own computer, that you can read in a few minutes, and that do one chore well.

Why small works

A small program on your own machine has some clear advantages.

- **It starts today.** You do not wait for a budget, a contract or a slot in someone else's plan. You run the wizard and the recipe works.
- **It fits your chore exactly.** A platform has to serve many people, so it offers what most of them need. A recipe is set up for your folders, your naming and your week.
- **You can read it.** Every program in this book is printed in full. When it does something you did not expect, you can look at the lines that did it.
- **It costs little.** jdBasic is free, the recipes are free, and they run on the computer you already have.
- **It goes away when you want.** If a recipe no longer helps, you switch it off and delete the folder. Nothing else depends on it.

There is also a less obvious advantage. Building or adapting a small program teaches you how your own work is structured. Many readers find that the most useful part of automating a chore is the moment they had to describe it clearly enough for a program, because that description showed them which steps were not needed at all.

When small is not enough

Small programs have limits, and it is worth knowing them before you reach them.

- **Shared data.** When several people change the same data, such as a customer list or a stock count, a script on one computer is the wrong place for it. That data belongs in a system everyone uses.

- **Rules you must follow.** Some work is governed by rules about records, retention, approvals or audit trails. A home made script rarely meets them, and it should not try. Ask before you automate anything in that area.
- **Many users.** A recipe that helps you can help your team, and Chapter 5 shows how to hand one out. When it is used by dozens of people, it needs support, updates and someone responsible. At that point it has become a product and should be treated as one.
- **Work that must not stop.** Your computer is switched off at night, updates restart it, and you go on holiday. If a chore must happen even when you are away, it needs a server, not a laptop.

None of these limits is a reason not to start. They are signs for when to talk to the people who run the larger systems. A working recipe is often the best way to start that conversation, because it shows exactly what you need.

Why jdBasic

The programs in this book are written in jdBasic, a language in the tradition of BASIC that was made for exactly this kind of work. It reads almost like English, which matters when you want to understand a program you did not write. It comes with the libraries that office work needs: Excel and Word files, PDF, mail, calendars, web pages, schedules. The same program runs in the interpreter while you try it and as a compiled program when you hand it to someone else. You do not need to know any of this to use the easy recipes, and Chapter 2 introduces as much of the language as the later chapters use.

Ethics and Boundaries

Automating your own work is your business, up to a point. The point is reached where your programs touch your employer's systems, other people's data or other people's inboxes. This section is not legal advice, and it does not replace the rules of your workplace. It names the questions you should be able to answer before you switch a recipe on.

Your employer's rules

Many workplaces have rules about software you may install, about where company data may be stored and about scripts that run on their computers. Find out what they are. In some offices installing `jdBasic` on your own is fine; in others it needs approval from IT. The recipes run without administrator rights and store everything in your own folders, which makes the request easier, but the rules still apply.

If you are not sure, ask. A short mail that explains what the program does and where its files live is usually enough. Chapter 6 has a section on talking to IT and security, with the questions they tend to ask and honest answers to them.

Personal data

Several recipes handle data about people: contact lists, birthdays, calendars, mail addresses, hours worked. In Europe the General Data Protection Regulation sets rules for this, and other regions have similar laws. In plain words, the ideas behind them are simple:

- keep only the personal data you need for a clear purpose;
- keep it safe, on your own computer or on systems your employer approves, and not on random web services;
- do not keep it longer than you need it;
- do not use it for something the people would not expect.

The recipes are written with this in mind. They work on files you already have, they keep their results in your work folder, and none of them sends data to a service on the internet unless you configure one. Recipe E12, the birthday reminder, is a good example to think about. A list of your colleagues' birthdays, kept for sending them good wishes, is fine for most people. The same list, used to sort them by age, is not.

Mail on behalf of others

Some recipes prepare mail, and later recipes can send it. Be careful with any program that sends mail in your name, and even more careful with one that sends in someone else's name. Mail that looks personal but was produced by a program can feel like a deception when the receiver finds out. A few rules of thumb:

- prepare, check, then send, at least until you are sure the program does the right thing in every case;
- keep templates honest: a thank you that was written once and is sent many times is fine, but it should not pretend to be something else;
- never send mail from another person's account without their explicit agreement;
- keep a copy of everything a program sends.

Monitor yourself, not your colleagues

The time tracker of recipe E07, the meeting cost meter of Chapter 4 and the balance score of Chapter 5 measure work. Used on your own week, they help you see where your time goes and make better decisions about it. Used on other people without their knowledge, the same programs become surveillance, and in many countries that is illegal as well as corrosive to trust.

This book only ever measures the reader. When a team lead in Chapter 4 builds a summary of the team's hours, it is from data the team enters itself, for a purpose everyone knows. If you adapt a recipe to measure others, talk to them first and to whoever handles staff matters in your workplace.

Passwords

Some recipes need a password: the mail server for sending, a web page behind a login, a guest network. The rule is simple: a password does not belong in a program and not in a settings file that others can read. The recipes in this book ask for passwords when they need them and keep them in memory only. Chapter 5 shows the job server that asks once in the morning, and recipe X14 discusses safer ways to store secrets when a program must run unattended.

If you ever find a password written into one of your own programs, take it out and change the password. It may already have been seen by anyone who received a copy.

Watch out

A backup or a zip archive made by a recipe contains everything in the folders you chose. If those folders hold personal data, the archive does too, and it needs the same care as the original.

How This Book Works

The rest of the book is built around recipes: small programs, each solving one chore, each with its own pages that explain it. This section describes how they are organised so that you can find your way around.

Three levels

The recipes come in three levels, fifteen each.

Easy recipes, in Chapter 3, are switched on by the setup wizard. You answer a few questions, the wizard writes a settings file, copies the programs and plans when they run. You can read the programs and change their settings, but you do not have to touch the code. These recipes are for anyone who uses a computer at work.

Medium recipes, in Chapter 4, are programs of one or two hundred lines that you set up yourself. They work with documents, mail, calendars and data: invoices from a spreadsheet, letters from a template, a weekly status mail, a cleaned up export. They expect you to read the program, change a few lines and run it. Chapter 2 teaches enough of the language for that.

Expert recipes, in Chapter 5, are small systems: a job server that runs everything on time, an assistant that drafts replies to mail, a search over your documents, tools you compile and hand to colleagues. They show how to write programs that log what they do, retry when the network is slow, test themselves and fail in a way that tells you what to fix.

You do not have to work through the levels in order. If an expert recipe solves the chore that costs you most, start there. Each recipe says what it needs before you start.

Three people

Three people appear throughout the book. They are not real, but their chores are.

Mia is a project assistant. She keeps documents, meetings and hours in order for three projects, and her day is full of small tasks at the computer. She has never programmed and does not intend to. The easy recipes are hers, and you have already seen her week.

Jonas is a team lead in a sales office. He lives in spreadsheets and in his inbox, writes the same kind of report every week, and collects numbers from six colleagues every month. He writes Excel formulas with confidence and has recorded a macro or two. The medium recipes start from his chores.

Priya is the IT generalist of a small company. She looks after the computers, the network and the accounts, and people come to her when something needs to work every day without anyone thinking about it. She programs when she has to. The expert recipes are the tools she builds for herself and for the others.

When a recipe begins with one of them, the story is short. It is there to make the chore concrete, so that you can decide quickly whether it is yours too.

The parts of a recipe

Every recipe follows the same order, so that you can skim it and find what you need.

1. **The chore** describes what the work costs now, usually in a short scene with one of the three people.
2. **What you get** shows the result in a picture or one sentence.
3. **Before you start** lists what must be in place.
4. **The program** prints the code, exactly as it was tested.
5. **How it works** walks through the program step by step.
6. **Run it** gives the command and what it prints.
7. **Schedule it** explains how the program runs without you.
8. **Make it yours** lists the settings and the lines you can change.
9. **When it goes wrong** covers the problems you are likely to meet.
10. **Balance dividend** says how much time the recipe gives back.

The programs are not typed into the book by hand. Each listing is taken from the file that was tested when the book was built, so what you read is what runs. The tests are in the same folders as the programs, and you can run them yourself.

What you need

For the easy recipes you need a Windows computer, the jdBasic release described in Chapter 2 and about fifteen minutes for the wizard. For the medium recipes you need a text editor and the patience to read a program from top to bottom once. For the expert recipes it helps to have written a program before, in any language. Some of them need access to a mail server, a web service or an AI model, and they say so at the start.

Starting Without Getting Stuck

Many readers will finish this chapter, look at a long audit and feel that there is too much to do. That feeling is the enemy of the whole idea. The way around it is to start with one chore, live with its program for a week, and only then add the next.

Pick one chore

From your audit, choose the line that meets all three of these conditions:

- it happens at least a few times a week;
- a mistake would be harmless and easy to undo;
- there is a recipe for it in Chapter 3.

For most people this is the Downloads folder, the renaming of scans or the morning routine of opening the same programs. Do not pick the largest chore first. Pick the safest one, so that your first experience with a program doing your work is a calm one.

The first week with a recipe

Run the recipe with its dry run for a day or two before you let it act. The dry run prints what the program would do and changes nothing. Read the list. You will find one of three things: everything is right, some files are treated in a way you did not expect, or the program finds nothing at all because a setting points at the wrong folder. All three are useful. In the second and third case, change a setting and run the dry run again.

When the list looks right, let the recipe act and schedule it. For the rest of the week, notice how often you would have done the chore by hand. That number is your first real entry for the ledger, and it is almost always different from the estimate in your audit.

At the end of the week, decide whether to keep the recipe. If yes, pick the next line. If not, switch it off without regret and write down why. A recipe that did not help tells you something about your work too.

Common objections

"I am not technical." The easy recipes do not ask you to be. You answer questions in a window, and the programs run on their own. If you want to change something, the settings are in one file with plain names. Chapter 2 shows the few lines of the language you may want to read, but nothing in Chapter 3 requires them.

"It is faster to do it by hand." For a single time, almost always. The program is faster over a month. If the setup really takes longer than the chore costs in a year, the automation test has already told you to keep it by hand, and that is a fine result.

"IT will not allow it." Perhaps. Ask with a clear description of what you want to run, where its files are and what it touches. Many IT departments are glad when someone asks first. If the answer is no, the audit and the ledger are still worth keeping, because they help you make the case for a tool that IT does approve.

"What if it breaks while I am away?" The recipes in this book fail in a visible way: they print a message and write it to a log, and they leave your files where they were. When you come back, the worst case is a chore that did not happen and is waiting for you as it used to. Chapter 6 describes a short monthly check that catches most problems before they matter.

"My work is too varied to automate." Some of it is. The audit usually shows that the varied part is the valuable part, and that it is surrounded by repeated chores that are not varied at all. The goal is to clear those away so that the varied work gets more of your attention.

"I will lose the skill." For chores like sorting files, nobody misses the skill. For anything that matters, the programs in this book are printed in full and explained line by line. Reading them is a better way to understand the chore

than repeating it, because the program has to spell out every step you used to do without thinking.

How long it takes

A rough picture from the recipes in Chapter 3: running the wizard takes about fifteen minutes. Each easy recipe needs a day or two of dry runs and a few minutes of settings. After a month most readers have four to six recipes running and have stopped thinking about the chores they took over. The medium recipes take an evening each, the expert ones a weekend. None of this has to happen at once. The ledger will show you the pace that suits you.

The Life Balance Ledger

The last tool of this chapter is the simplest one in the book. It is a table that you keep for yourself, once a week, for as long as you use the recipes. It answers two questions that otherwise get lost: how much time did the programs give back, and what did that time go to?

The table

Keep the ledger wherever you keep things you look at every week: a sheet in a spreadsheet, a page in a notebook or a text file in your work folder. It has four columns.

Week	Recipe	Minutes saved	What the time went to
------	--------	---------------	-----------------------

- **Week:** the date of the Monday, so the rows sort by date.
- **Recipe:** the recipe, such as E01 Downloads Butler, or "all" if you prefer one line per week.
- **Minutes saved:** your estimate for that week. Each recipe gives a starting value in its Balance dividend box; replace it with your own number once you know it.
- **What the time went to:** one honest line. "Left at five on Tuesday", "Read the first half of the course", or "I do not know", which is also useful to know.

Here is the start of Mia's ledger after her first month with the easy recipes.

Week	Recipe	Minutes saved	What the time went to
2026-10-05	E01 Downloads Butler	30	Nothing planned yet, it disappeared
2026-10-05	E03 Batch Renamer	20	Same
2026-10-12	E01, E03, E07	85	Left on time on Wednesday and Friday
2026-10-19	E01, E03, E07, E08	110	Thursday morning hour for the archive cleanup
2026-10-26	E01, E03, E07, E08, E13	125	Thursday hour again; one long lunch

The first line is the most common first line of all. The time came back and went nowhere in particular, because nothing was waiting for it. Once Mia wrote down in advance that the minutes were for leaving on time, they stayed hers.

Adding it up

If you keep the ledger as a text file, a few lines of `jdBasic` can add it up for you. Here is a small example with the minutes of four weeks. You will understand every line of it after Chapter 2; for now, it shows how little a program needs to be useful.

```
DIM weeks = ["10-05", "10-12", "10-19", "10-26"]
DIM minutes = [50, 85, 110, 125]
DIM total = SUM(minutes)
PRINT "Minutes saved in four weeks: "; total
PRINT "That is "; ROUND(total / 60, 1); " hours"
PRINT "Average per week: "; total / LEN(weeks); " minutes"
```

```
Minutes saved in four weeks: 370
That is 6.2 hours
Average per week: 92.5 minutes
```

How the book uses the ledger

Every recipe ends with a box that looks like this:

Balance dividend

About 15 minutes a week of tidying and searching, and the half hour once a month that the folder used to cost you.

The number in the box is an estimate for a typical week. Yours will be different, higher for some recipes and lower for others. Use the box as the first entry in your ledger and correct it after a few weeks.

Chapter 6 comes back to the ledger after three months. By then you will have enough rows to see patterns: which recipes really help, which ones you could switch off, and whether the time you won went where you wanted it to go. Recipe X15 in Chapter 5 even turns the ledger, your calendar and your timesheet into a weekly balance score, for those who like a number. The paper version works just as well.

Before you turn the page

You now have three things that the rest of the book builds on.

- Your audit, with the chores of one week and the minutes they cost.
- The automation test, which tells you which chores to hand over and which to keep.
- An empty ledger, with a line at the top that says what the saved time is for.

Chapter 2 installs jdBasic, runs the setup wizard and switches on the first recipes. Pick the two or three lines of your audit with the most minutes and look for the matching recipes in the list at the start of Chapter 3. That is where the first time comes back.

Chapter 2: Your Toolbox

This chapter puts the tools on your computer and shows you how to use them. It starts with what needs no programming at all: installing `jdBasic` and running the setup wizard, which switches on the fifteen automations of Chapter 3. Then it teaches the small part of `jdBasic` the recipes are written in, the libraries they lean on, and how a program runs on its own at a fixed time.

If you only want the easy recipes, the first two sections are enough. Come back for the rest when you want a recipe to do one thing differently, or when you reach Chapter 4.

Installing `jdBasic`

`jdBasic` is a single program, `jdbasic.exe`, with a few libraries next to it. It needs no administrator rights, no runtime to install first and no account anywhere.

The book uses the release with Windows forms, because the setup wizard opens a window. You find it on the releases page of the project, github.com/AtomijD/jdBasic/releases. Download the archive, unpack it into a folder of your own, for example `C:\Users\mia\jdBasic`, and add that folder to the `PATH` so that Windows finds `jdbasic` from any command prompt:

1. Open the Start menu and type *environment variables*.
2. Choose *Edit environment variables for your account*.
3. Select *Path*, choose *Edit*, then *New*, and paste the folder.
4. Close every command prompt that was open; new ones see the change.

Check it in a new command prompt:


```
jdbasic --version
```

The answer names the version and the parts the release was built with. *Forms* must be among them for the setup wizard.

Watch out

Some offices do not allow programs outside the folders IT installs. If Windows refuses to start `jdbasic.exe`, ask IT before you look for a way around it. *Talking to IT and Security* in Chapter 6 is about that conversation.

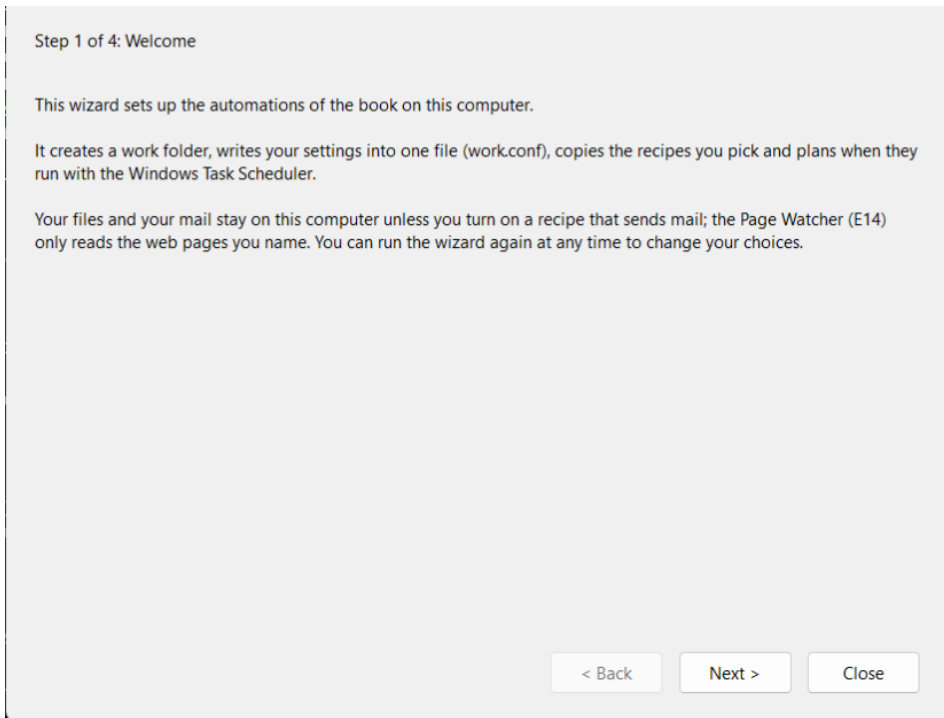
The recipes of the book come with the book: the folder `recipes` next to the manuscript, or the download that goes with your copy. Unpack it anywhere; the wizard copies what you choose into your work folder.

The Setup Wizard

The wizard is the reason the easy level needs no programming. Start it from the folder of the book's files:

```
jdbasic wizard\setup_wizard.jdb
```

It has four pages. You can go back and forth between them as often as you like; nothing happens on your computer until you press *Set up now* on the last page.



The first page says what the wizard will do and what it never does

The first page explains the plan: a work folder, one settings file, the recipes you choose, and a schedule for each of them. It also says what the wizard never does: no recipe of the easy level sends your files or your mail anywhere. The only one that uses the network at all is the Page Watcher (E14), which reads the web pages you name.

Step 2 of 4: About you

Work folder

The jdbasic program

Your name

Your mail address

Working day starts at

Working day ends at

Mail server (optional)

Mail user name

< Back Next > Close

About you: the work folder, the jdbasic program, your name, your working hours and an optional mail server

The second page asks about you. The work folder is where your settings, the copied recipes and their logs live; the suggestion inside Documents is a good one. The wizard finds jdbasic on its own when it is on the PATH. Your name and mail address go into the mail templates of E08, and the working hours decide when E05 reminds you of breaks and when E06 asks its three questions at the end of the day. The mail server is optional and only matters for the recipes of Chapter 4 that send mail; leave it empty for now. The wizard never asks for a mail password: recipes that send mail ask for it when you run them yourself, and nothing writes it down.

Step 3 of 4: Your automations

[x] E01 Downloads Butler (15 min)

[x] E02 Morning Launcher (10 min)

[x] E03 Batch Renamer (10 min)

[x] E04 One-Click Backup (15 min)

[x] E05 Break Reminder (0 min)

[x] E06 Shutdown Ritual (20 min)

[x] E07 Time Tracker (30 min)

[x] E08 Mail Templates (25 min)

[x] E09 Meeting Notes Starter (20 min)

[x] E10 Duplicate Finder (10 min)

[x] E11 Disk Space Report (5 min)

[x] E12 Birthday Reminder (10 min)

[x] E13 Printable Week (10 min)

[x] E14 Page Watcher (20 min)

[x] E15 Guest Wi-Fi Card (5 min)

[] M01 Invoice Generator (10 min)

[] M02 Report Merger (15 min)

[] M03 Mail Merge (5 min)

[] M04 Receipt Sorter (15 min)

[] M05 Shift and Vacation Planner (30 min)

[] M06 Friday Status Mail (45 min)

[] M07 Inbox Unpacker (30 min)

☒ Use this recipe

Sorts the Downloads folder into folders by kind and month

Folder to tidy
~/Downloads

Folder to sort into
~/Downloads/Sorted

Leave files younger than (hours)
24

When to run (daily 18:30, weekdays 08:30, every 50 minutes, manual)
daily 18:30

< Back

Next >

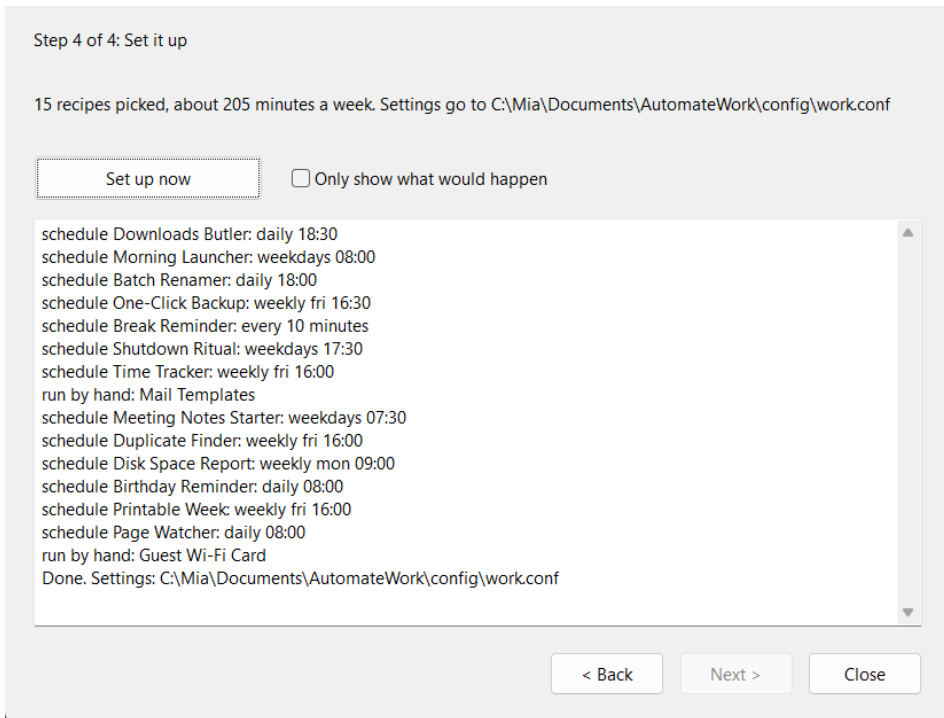
Close

Your automations: every recipe with the minutes it saves, its settings and its schedule

The third page is the heart of the wizard. On the left are the recipes with the minutes each one saves in a normal week; [x] marks the ones that are switched on. At first these are the fifteen easy ones; the recipes of Chapter 4 are listed below them and stay off until you switch them on. Click a recipe to see its settings on the right. Each setting has a label and a sensible default, such as the folder the Downloads Butler tidies. At the bottom is the schedule:

- daily 18:30 runs it every day at half past six in the evening,
- weekdays 08:30 from Monday to Friday,
- weekly fri 16:00 once a week,
- monthly 1 08:00 on the first of every month, or any day up to the 28th,
- every 50 minutes again and again while you are logged in,
- manual (or an empty field) never on its own; you start it yourself.

Leave a recipe switched off if you are not sure about it. You can run the wizard again at any time, and it remembers everything you chose.



Set it up: the plan, the button, and what happened

The last page sums up what you picked and how many minutes a week that promises. *Only show what would happen* lets you see the plan first. *Set up now* creates the work folder, writes the settings file, copies the recipes, and adds one task per recipe to the Windows Task Scheduler, in a folder of its own called *AutomateWork*.

Try this

Run the wizard with *Only show what would happen* ticked first. It lists every schedule without touching anything. When the list looks right, untick it and set up for real.

The settings file

Everything you chose ends up in one text file, `work.conf`, in the `config` folder of your work folder. It is written in TOML, a format for settings that reads like a list of name and value pairs grouped under headings in brackets:

```
[user]
name = "Mia Example"
day_start = "08:30"
day_end = "17:30"

[downloads_butler]
source = "~/Downloads"
target = "~/Downloads/Sorted"
min_age_hours = 24
```

Every recipe reads its own part, the one under its heading. ~ stands for your home folder. Text goes in double quotes, numbers and true or false do not. Inside double quotes a backslash starts a special character, so write folders with forward slashes, "C:/Users/mia/Scans", which Windows understands as well. You can change this file with any text editor; the next run of a recipe uses the new values, and the next run of the wizard shows them.

The wizard without a window

On a computer without the forms release, or if you prefer the command prompt, the same wizard runs in the console. It asks the same questions one after the other, with the current answer in brackets; Enter keeps it:

```
jdbasic wizard\setup_console.jdb
```

Two switches help later on. --doctor checks an existing setup and lists what is missing, a moved folder or a deleted task, for example. --uninstall removes the scheduled tasks and leaves your files and your settings alone.

Your First Program

A jdBasic program is a text file that ends in .jdb. Open Notepad, type the four lines below, save them as hello.jdb in your work folder, and run them from a command prompt in that folder with jdbasic hello.jdb. The second box shows what appears:

```
DIM who$ = "Mia"
DIM day = CDATE("2026-10-05 08:30:00")
PRINT "Good morning, " + who$ + "."
PRINT "Today is " + FORMAT_DATE(day, "%A, %d %B %Y") + "."
```

```
Good morning, Mia.  
Today is Monday, 05 October 2026.
```

That is the whole cycle: write lines, save, run, read what it printed. Every program in this book works this way, the large ones included. Change "Mia" to your own name and run it again. Then replace the fixed date with `NOW()`, the date and time of the moment the program runs, and it greets you with today.

The next section explains every word of these four lines, and the rest of what the recipes use.

jdBasic for Office People

This part of the chapter teaches you enough jdBasic to read every program in Chapter 3 and to change it. It does not try to teach everything. It teaches the pieces the recipes use, in the order you meet them, and each piece comes with a small example you can type in and run.

You do not need to work through it in one sitting. Read the first few sections, set up a recipe or two, and come back when a listing shows you something you have not seen yet. The recipes point back here.

What a program is

A program is a text file with instructions, one per line, that the computer carries out from the top to the bottom. jdBasic programs end in `.jdb`. You write them in any text editor: Notepad works, Visual Studio Code is more comfortable because it colours the code.

To run a program, open a command window in the folder where the file is and type `jdbasic` followed by the file name:

```
jdbasic hello.jdb
```

Here is the shortest useful program. It prints one line:

```
PRINT "Good morning, Mia."
```

```
Good morning, Mia.
```

Every example in this chapter looks like this: first the program, then, in a lighter box, exactly what it prints. Every one of them was run when this book was built, and the build would have stopped if a single character of the output had been different from what you see here.

Try this

Save the line above as `hello.jdb`, change the name to yours, and run it. If the window says it does not know `jdbasic`, the installation in the first part of this chapter is not finished yet.

PRINT

PRINT writes text to the screen. Text goes between double quotes. A semicolon joins several things on one line with nothing in between, a comma puts a space between them, and PRINT alone writes an empty line.

```
PRINT "Invoices: "; 12
PRINT "Paid", "Open", "Late"
PRINT
PRINT "done"
```

```
Invoices: 12
Paid Open Late

done
```

A semicolon at the very end of a line keeps the next PRINT on the same line. You will see that in loops further down.

Variables and DIM

A variable is a name for a value that the program keeps while it runs. You create one with DIM and give it a value with =. After that you can use the name wherever you would use the value, and you can give it a new value.


```
DIM invoices = 12
DIM paid = 9
PRINT "Open: "; invoices - paid
invoices = invoices + 3
PRINT "After the post came: "; invoices
```

```
Open: 3
After the post came: 15
```

A name ending in \$ holds text. This is an old BASIC habit, and the recipes keep it because it makes a listing easier to read: when you see `folder$`, you know it is text without looking further.

```
DIM person$ = "Mia"
DIM greeting$ = "Hello, " + person$
PRINT greeting$
```

```
Hello, Mia
```

Names may contain letters, digits and the underscore, and capital and small letters count as the same: `Total` and `total` are one variable. A few names belong to `jdBasic` itself. Words of the language such as `TEXT`, `LINE` or `STEP` cannot be names at all, and the name of a builtin such as `COUNT` or `SORT` cannot be the name of a function of your own; a program that tries stops with an error. Appendix A lists them. The recipes use names such as `n_files` or `line_text$` instead.

Typing mistakes and **OPTION "EXPLICIT"**

A variable that was never created has no value, which `jdBasic` calls `NONE`. In a calculation, `NONE` counts as 0. That is convenient while you try things out and treacherous in a program you rely on: a typing mistake in a name gives no error, only a wrong result.

```
DIM total = 10
PRINT totl + 1
PRINT TYPEOF(totl)
```

```
1
NONE
```

The first line of output is 1, not 11, because `totl` is not `total`. The line `OPTION "EXPLICIT"` at the top of a program asks `jdBasic` to refuse every name that was not created with `DIM`. The program then stops before it runs and names the line:

```
OPTION "EXPLICIT"
DIM total = 10
PRINT totl + 1
```

```
Error #20: Undeclared variable 'TOTL' (OPTION "EXPLICIT") at line 3
```

The recipes are written so that they would pass this check, and when you start a program of your own, it is worth writing `OPTION "EXPLICIT"` as its first line. You will thank yourself the first time it catches a mistake that would otherwise have moved files to the wrong folder.

Numbers

`jdBasic` calculates with `+`, `-`, `*` and `/`. Two more operators help with whole numbers: `\` divides and drops the remainder, `MOD` gives the remainder. `^` raises to a power.

```
PRINT 7 / 2, 7 \ 2, 7 MOD 2, 2 ^ 10
PRINT 90 / 60, 90 \ 60, 90 MOD 60
```

```
3.5 3 1 1024
1.5 1 30
```

The second line is the start of every time calculation in this book: 90 minutes are one hour and thirty minutes. The time tracker in E07 works this way.

Numbers with a fractional part sometimes do not come out exactly as you would write them, because the computer stores them in binary. `jdBasic` rounds the printed value, so in practice you rarely notice:

```
PRINT 0.1 + 0.2
PRINT ROUND(2 / 3, 2)
```

```
0.3
0.67
```

Text and numbers together

+ between two numbers adds them. Between text and anything else it joins them into one longer text:

```
DIM files = 3
PRINT "You have " + files + " new files."
PRINT "1" + "1", 1 + 1
```

```
You have 3 new files.
11 2
```

When a program reads a number from a file or from the keyboard, it arrives as text. VAL and CINT turn text into a number. CINT gives a whole number and stops with an error when the text is not a number at all, which is usually what you want; VAL answers 0 instead. STR\$ goes the other way.

```
PRINT VAL("3.5") + 1, CINT("12") + 1
PRINT STR$(42) + "!"
PRINT VAL("twelve")
```

```
4.5 13
42!
0
```

Text with values inside

Joining text with + gets long when a sentence holds several values. A text that starts with \$ before the quote may hold expressions between double curly braces, and jdBasic puts in their values:

```
DIM n = 3
DIM folder$ = "Downloads"
PRINT $"{{n}} files are waiting in {{folder$}}."
PRINT $"That is {{n * 2}} minutes of sorting."
```

```
3 files are waiting in Downloads.
That is 6 minutes of sorting.
```

For numbers with a fixed number of decimals, `FORMAT$` takes a pattern with `{}` where a value goes. `{:.2f}` means two digits after the point, `{:>8}` means right aligned in eight places:

```
PRINT FORMAT$("Total: {:.2f} EUR", 209.8)
PRINT FORMAT$("[{:>8}]", "right")
PRINT FORMAT$("{} of {} done", 4, 15)
```

```
Total: 209.80 EUR
[   right]
4 of 15 done
```

Working with text

Most of what the recipes do is looking at names of files and lines of files, so text functions appear on almost every page of Chapter 3.

`LEN` gives the length of a text. `LEFT$` and `RIGHT$` take characters from the start and the end. `MID$` takes characters from the middle.

```
DIM file$ = "2026-10-01 Offer.pdf"
PRINT LEN(file$)
PRINT LEFT$(file$, 10)
PRINT RIGHT$(file$, 4)
PRINT MID$(file$, 11, 5)
```

```
20
2026-10-01
.pdf
Offer
```

Watch out

MID\$ counts from 0, not from 1: the first character is at position 0. MID\$(file\$, 11, 5) starts at the twelfth character. Many other BASICs count from 1, so code copied from elsewhere may be off by one.

INSTR finds one text inside another and answers the position where it starts, again counted from 0. When the text is not there, it answers -1.

```
DIM file$ = "2026-10-01 Offer.pdf"
PRINT INSTR(file$, "Offer")
PRINT INSTR(file$, "Invoice")
IF INSTR(file$, "Offer") >= 0 THEN PRINT "an offer"
```

```
11
-1
an offer
```

Often you only want to know whether a text appears at all. IN answers that directly, and STARTSWITH and ENDSWITH look at the two ends:

```
DIM file$ = "2026-10-01 Offer.pdf"
PRINT "Offer" IN file$
PRINT STARTSWITH(file$, "2026"), ENDSWITH(file$, ".pdf")
```

```
TRUE
TRUE TRUE
```

UCASE\$ and LCASE\$ change capital and small letters; they help when you compare text that people typed, since "PDF" and "pdf" are different texts. TRIM\$ removes spaces at both ends. REPLACE\$ changes every occurrence of one text into another.

```
PRINT UCASE$("pdf"), LCASE$("Report.PDF")
PRINT "[" + TRIM$(" Mia ") + "]"
PRINT REPLACE$("2026-10-01", "-", ".")
PRINT LCASE$("Offer.PDF") = "offer.pdf"
```

```
PDF report.pdf
[Mia]
2026.10.01
TRUE
```

SPLIT cuts a text into a list of pieces at a separator, and JOIN puts a list back together. Both appear wherever a recipe reads a line with several values in it:

```
DIM line_text$ = "Mia;mia@example.com;Projects"
DIM parts = SPLIT(line_text$, ";")
PRINT parts
PRINT parts[1]
PRINT JOIN(parts, " / ")
```

```
[Mia, mia@example.com, Projects]
mia@example.com
Mia / mia@example.com / Projects
```

Umlauts and LEN

LEN counts bytes, and an umlaut takes two bytes in the UTF-8 text `jdBasic` uses. `LEN("Müller")` is 7. When you need the number of characters, write `LEN(-name$)`: the minus in front of a text turns it into a list of its characters. `LEFT$` and `MID$` count bytes too, so cut names with umlauts at a space or a known position, not in the middle of a word.

```
DIM name_text$ = "Müller"
PRINT LEN(name_text$), LEN(-name_text$)
PRINT -name_text$
```

```
7 6
[M, ü, l, l, e, r]
```

Lists

A list, called an *array* in programming, holds several values in one variable, in order. You write it in square brackets. `LEN` gives the number of entries, and a number in square brackets after the name picks one entry. The first entry has the number 0.

```
DIM files = ["offer.pdf", "photo.jpg", "notes.txt"]
PRINT LEN(files)
PRINT files[0]
PRINT files[2]
PRINT files
```

```
3
offer.pdf
notes.txt
[offer.pdf, photo.jpg, notes.txt]
```

PUSH adds an entry at the end. APPEND answers a new, longer list and leaves the old one as it is. SORT answers a sorted copy, in reverse order when you add TRUE:

```
DIM files = ["offer.pdf", "photo.jpg"]
PUSH(files, "budget.xlsx")
PRINT files
DIM longer = APPEND(files, "agenda.docx")
PRINT LEN(files), LEN(longer)
PRINT SORT(files)
PRINT SORT([3, 10, 7], TRUE)
```

```
[offer.pdf, photo.jpg, budget.xlsx]
3 4
[budget.xlsx, offer.pdf, photo.jpg]
[10, 7, 3]
```

IN works on lists as well, and answers whether a value is one of the entries. The Downloads Butler in E01 uses exactly this to decide what kind of file it has in front of it:

```
DIM images = ["jpg", "jpeg", "png", "heic"]
PRINT "png" IN images
PRINT "pdf" IN images
```

```
TRUE
FALSE
```

Maps

A map stores values under names, the way a phone book stores numbers under people. You write it in curly braces with "name": value pairs, and you read a value with the name in curly braces after the map. The names are called *keys*.

```
DIM kinds = {"pdf": "Documents", "jpg": "Images"}
PRINT kinds{"pdf"}
kinds{"zip"} = "Archives"
PRINT MAP.KEYS(kinds)
PRINT MAP.EXISTS(kinds, "zip"), MAP.EXISTS(kinds, "exe")
```

```
Documents
[pdf, jpg, zip]
TRUE FALSE
```

Reading a key that is not in the map gives `NONE`, `jdBasic`'s word for "no value". The operator `??` picks a default for that case, which keeps many programs short:

```
DIM kinds = {"pdf": "Documents", "jpg": "Images"}
PRINT kinds{"exe"} ?? "Other"
PRINT kinds{"pdf"} ?? "Other"
```

```
Other
Documents
```

The settings file of the book, `work.conf`, arrives in every recipe as a map of maps: the part `[downloads_butler]` becomes one map inside it, with the settings as its keys. That is why the recipes read their settings with names such as `"downloads_butler.source"`.

Decisions: IF

`IF` runs a piece of the program only when a condition holds. The condition compares values: `=` equal, `<>` not equal, `<`, `>`, `<=` and `>=`. A short decision fits on one line with `THEN`; a longer one has `ELSEIF` and `ELSE` branches and ends with `ENDIF`.


```

DIM size_mb = 1500
IF size_mb > 1000 THEN
    PRINT "a large file"
ELSEIF size_mb > 100 THEN
    PRINT "a medium file"
ELSE
    PRINT "a small file"
ENDIF
IF size_mb > 1000 THEN PRINT "consider moving it to the archive"

```

```

a large file
consider moving it to the archive

```

Comparisons answer TRUE or FALSE, and you can print those values or keep them in a variable. AND and OR combine two conditions, NOT turns one around:

```

DIM hour = 18
DIM weekday = TRUE
PRINT hour >= 17 AND weekday
PRINT hour < 8 OR NOT weekday
PRINT "Mia" = "mia"

```

```

TRUE
FALSE
FALSE

```

The last line shows that comparing text is exact: capital and small letters differ. Compare LCASE\$ of both sides when that should not matter.

ANDALSO and ORELSE work like AND and OR but stop as soon as the answer is clear. That matters when the second condition only makes sense if the first one holds:

```

DIM kinds = {"pdf": "Documents"}
DIM ext$ = "zip"
IF MAP.EXISTS(kinds, ext$) ANDALSO kinds{ext$} = "Documents" THEN
    PRINT "a document"
ELSE
    PRINT "something else"
ENDIF

```

```
something else
```

Repeating: loops

FOR ... NEXT counts. The variable takes every value from the first number to the second, and the lines in between run once for each:

```
DIM k = 0
FOR k = 1 TO 3
    PRINT "Reminder number "; k
NEXT k
FOR k = 10 TO 0 STEP -5
    PRINT k; " ";
NEXT k
PRINT
```

```
Reminder number 1
Reminder number 2
Reminder number 3
10 5 0
```

FOR EACH walks through a list, one entry after the other, and is the loop you will see most often in Chapter 3. With two names it gives the position as well:

```
FOR EACH file$ IN ["offer.pdf", "photo.jpg"]
    PRINT "found "; file$
NEXT
FOR EACH i, file$ IN ["offer.pdf", "photo.jpg"]
    PRINT i; ": "; file$
NEXT
```

```
found offer.pdf
found photo.jpg
0: offer.pdf
1: photo.jpg
```

On a map, FOR EACH with two names gives each key with its value, in the order the keys were added:

```
DIM hours = {"Project A": 12, "Project B": 7.5}
FOR EACH project$, h IN hours
    PRINT project$; ": "; h; " hours"
NEXT
```

```
Project A: 12 hours
Project B: 7.5 hours
```

DO WHILE ... LOOP repeats as long as a condition holds, and is the loop for "until there is nothing left to do":

```
DIM remaining = 3
DO WHILE remaining > 0
    PRINT "files left: "; remaining
    remaining = remaining - 1
LOOP
PRINT "all sorted"
```

```
files left: 3
files left: 2
files left: 1
all sorted
```

Watch out

A DO WHILE loop whose condition never becomes false runs for ever. If a program of yours seems to hang, press Ctrl+C in its window to stop it, and check that something inside the loop changes the condition.

Whole lists at once

Many loops do one of three simple things: they turn every entry of a list into something else, they keep the entries that pass a test, or they add numbers up. jdBASIC does each of these for the whole list in one line, and the line says what it does without a loop around it.

Arithmetic and the functions for numbers work on a whole list. IOTA(*n*) makes the list of the numbers from 1 to *n*:

```
DIM hours = [7.5, 8, 6.25, 9]
PRINT SUM(hours)
PRINT MAX(hours) - MIN(hours)
PRINT hours * 60
PRINT IOTA(5)
```

```
30.75
2.75
[450, 480, 375, 540]
[1, 2, 3, 4, 5]
```

SELECT turns every entry into something else, and FILTER keeps the entries that pass a test. What happens to each entry is written as a LAMBDA: a small function without a name, with its parameter before the arrow and its result after it. LAMBDA f\$ -> UCASE\$(f\$) reads as "take f\$ and answer it in capitals":

```
DIM files = ["offer.pdf", "photo.jpg", "budget.xlsx", "notes.pdf"]
PRINT SELECT(LAMBDA f$ -> UCASE$(f$), files)
PRINT FILTER(LAMBDA f$ -> ENDSWITH(f$, ".pdf"), files)
PRINT LEN(FILTER(LAMBDA f$ -> ENDSWITH(f$, ".pdf"), files))
```

```
[OFFER.PDF, PHOTO.JPG, BUDGET.XLSX, NOTES.PDF]
[offer.pdf, notes.pdf]
2
```

When the test is longer than one expression, write it as a function of its own and hand it over with @ after its name:

```
FUNC IsWorkday(day$)
    RETURN NOT (day$ IN ["Sat", "Sun"])
ENDFUNC
DIM days = ["Mon", "Tue", "Sat", "Sun", "Fri"]
PRINT FILTER(IsWorkday@, days)
```

```
[Mon, Tue, Fri]
```

GRADE answers the positions of the entries in sorted order. With that list in the brackets, another list comes out sorted the same way, so names can be sorted by their hours. UNIQUE drops repeated entries:

```
DIM names = ["Mia", "Jonas", "Lena"]
DIM hours = [6.5, 9, 7.25]
PRINT GRADE(hours)
PRINT names[GRADE(hours)]
PRINT UNIQUE(["pdf", "jpg", "pdf", "docx", "jpg"])
```

```
[0, 2, 1]
[Mia, Lena, Jonas]
[pdf, jpg, docx]
```

The same works on a list of maps, which is how the recipes hold rows of a sheet or entries of a log:

```
DIM team = [{"name": "Mia", "hours": 6.5}, _
            {"name": "Jonas", "hours": 9}]
PRINT SELECT(LAMBDA p -> p{"name"}, team)
PRINT SUM(SELECT(LAMBDA p -> p{"hours"}, team))
DIM busy = FILTER(LAMBDA p -> p{"hours"} > 8, team)
PRINT busy[0>{"name"}
```

```
[Mia, Jonas]
15.5
Jonas
```

REDUCE folds a list into one value, starting from the value after the list. Here it turns hours into minutes and adds them up in one go:

```
PRINT REDUCE(LAMBDA total, h -> total + h * 60, [1.5, 2], 0)
```

```
210
```

A loop is still the right tool when each step does something outside the list: it reads or writes a file, prints a line, starts a program, or stops early once it found what it looked for. The recipes use both: the list functions to decide, a loop to act on the decision.

Dates and times

Half of the recipes in Chapter 3 ask what day it is: the break reminder wants to know whether the working day has started, the shutdown ritual whether it is Friday, the time tracker how many hours lie between start and stop, and the birthday reminder how many days remain. `jdBasic` keeps a point in time as a date value. `NOW()` gives the present one, and `CDATE` makes one from a text in the form year, month, day and optionally hours, minutes and seconds.

`FORMAT_DATE` turns a date into text with a pattern. `%Y` stands for the year, `%m` for the month, `%d` for the day, `%H` and `%M` for hours and minutes, `%A` and `%B` for the names of the weekday and the month:

```
DIM start = CDATE("2026-10-01 08:30:00")
PRINT FORMAT_DATE(start, "%Y-%m-%d %H:%M")
PRINT FORMAT_DATE(start, "%A, %d %B %Y")
PRINT YEAR(start), MONTH(start), DAY(start)
```

```
2026-10-01 08:30
Thursday, 01 October 2026
2026 10 1
```

`DATEADD` moves a date by a number of units: "D" for days, "H" for hours, "N" for minutes (the M is taken by months). `DATEDIFF` answers how many units lie between two dates. Note the order of the values: the unit comes first, then the number or the earlier date.

```
DIM start = CDATE("2026-10-01 08:30:00")
PRINT FORMAT_DATE(DATEADD("D", 3, start), "%Y-%m-%d")
PRINT FORMAT_DATE(DATEADD("H", 9, start), "%H:%M")
DIM finish = CDATE("2026-10-01 17:00:00")
PRINT DATEDIFF("H", start, finish)
PRINT DATEDIFF("N", start, CDATE("2026-10-01 09:15:00"))
PRINT DATEDIFF("D", CDATE("2026-10-01"), CDATE("2026-12-24"))
```

```
2026-10-04
17:30
8.5
45
84
```

The working day in the example is 8.5 hours long, and Christmas is 84 days away. The weekday comes out of `FORMAT_DATE` as well: `%w` gives a digit, 0 for Sunday, 1 for Monday and so on up to 6 for Saturday. The recipes use it to tell weekdays from weekends.

```
DIM friday = CDATE("2026-10-02 16:00:00")
DIM wd = VAL(FORMAT_DATE(friday, "%w"))
PRINT wd
IF wd = 5 THEN PRINT "Friday: time for the week summary"
IF wd = 0 OR wd = 6 THEN PRINT "weekend"
```

```
5
Friday: time for the week summary
```

The examples here use fixed dates so that they always print the same. In the recipes the same lines work on `NOW()`, and the tests of each recipe hand in a fixed date to check what the program would do on a given day, such as a Friday or the last day of February.

Your own functions: FUNC and SUB

When a program does the same thing in several places, you give that thing a name. A `FUNC` takes values, called *parameters*, calculates something and hands back the result with `RETURN`. A `SUB` does something and hands back nothing. A parameter may have a default value, which is used when the caller leaves it out.

```
FUNC Minutes(hours)
    RETURN hours * 60
ENDFUNC

FUNC Greeting$(person$, polite = TRUE)
    IF polite THEN RETURN "Dear " + person$ + ", "
    RETURN "Hi " + person$ + ", "
ENDFUNC

SUB Banner(title$)
    PRINT "== " + title$ + " =="
ENDSUB

Banner("Monday")
PRINT Minutes(1.5)
```

```
PRINT Greeting$("Ms Miller")
PRINT Greeting$("Jonas", FALSE)
```

```
== Monday ==
90
Dear Ms Miller,
Hi Jonas,
```

As with variables, a function whose name ends in \$ hands back text. Variables created inside a function belong to it and disappear when it returns, so two functions can each have their own `k` without getting in each other's way.

Every recipe in Chapter 3 consists mostly of functions: `KIND$` in E01 decides the kind of a file, `PLAN` decides what to move, and `APPLY` moves it. Functions with clear names are what makes a program readable a year later.

Modules: IMPORT

A module is a file of functions that other programs use. The book comes with modules for Word, Excel and PDF files, for mail, calendars, settings and many other things, and every recipe has its own module next to its program. `IMPORT` makes a module's functions available; you call them with the module's name, a dot and the function's name:

```
IMPORT CONF
DIM q$ = CHR$(34)
DIM cfg = CONF.TOML_TEXT("[mail]" + CHR$(10) + "to = " + q$ + _
    "mia@example.com" + q$)
PRINT CONF.GET(cfg, "mail.to", "nobody")
PRINT CONF.GET(cfg, "mail.cc", "nobody")
```

```
mia@example.com
nobody
```

The `_` at the end of a line continues the instruction on the next one. The listings in this book use it to stay narrow enough for the page.

A file becomes a module by starting with `EXPORT MODULE` and its name, and by marking the functions others may call with `EXPORT`:


```
EXPORT MODULE BUTLER

EXPORT FUNC KIND$(file_name$, rules)
    ' ...
ENDFUNC
```

jdBasic looks for a module first in the folder of the program that imports it, then in a few standard places, among them the folder `.jdbasic\lib` in your home folder. The setup wizard puts the shared module of the recipes, `WORKCONF`, there, so every recipe finds it.

When something fails: TRY and CATCH

Some things go wrong in normal use: a file is missing, a setting holds text where a number belongs, a web page is not reachable. Without precautions the program stops with an error message. TRY marks the lines that may fail, and the lines after CATCH run instead when they do. ERRMSG\$ holds the message:

```
TRY
    DIM v = CINT("twelve")
    PRINT "a number"
CATCH
    PRINT "Problem: "; ERRMSG$
ENDTRY
TRY
    PRINT TXTREADER$("C:/no/such/folder/notes.txt")
CATCH
    PRINT "Problem: "; ERRMSG$
ENDTRY
PRINT "the program goes on"
```

```
Problem: CINT: "twelve" is not a number
Problem: TXTREADER$: Cannot open file: C:/no/such/folder/notes.txt
the program goes on
```

Your own program can report a problem the same way with THROW. The recipes do that when a setting is missing, so the message says what to do about it:

```
DIM source$ = ""
TRY
    IF source$ = "" THEN THROW "Set downloads_butler.source first."
CATCH
```

```
PRINT ERRMSG$
ENDTRY
```

```
Set downloads_butler.source first.
```

Comments

A line, or the rest of a line, that starts with an apostrophe is a comment. `jdBasic` ignores it; it is there for the person who reads the program. The recipes start every function with a comment that says what it does, in one or two sentences.

```
' The minutes a chore costs in a year, at so many minutes a week.
FUNC YearMinutes(per_week)
    RETURN per_week * 46 ' working weeks, without holidays
ENDFUNC
PRINT YearMinutes(15)
```

```
690
```

Money, hours and tidy columns

Reports are the reason many chores exist, and a report is mostly numbers in columns. Three tools cover what the recipes need. `ROUND` rounds to a number of decimals. `FORMAT$` with a width keeps columns straight: `{:<12}` fills to twelve places with the text on the left, `{:>8.2f}` puts a number with two decimals at the right of eight places. And `SUM` adds up a whole list at once.

```
DIM items = ["Paper", "Toner", "Delivery"]
DIM prices = [45.9, 123.8, 9]
DIM k = 0
FOR k = 0 TO LEN(items) - 1
    PRINT FORMAT$("{:<12}{:>8.2f}", items[k], prices[k])
NEXT k
PRINT FORMAT$("{:<12}{:>8.2f}", "Total", SUM(prices))
PRINT ROUND(178.7 * 0.19, 2)
```

Paper	45.90
Toner	123.80
Delivery	9.00

Total	178.70
33.95	

Times work the same way once you count in minutes. A time tracker that stores minutes can print them as hours and minutes with `\` and `MOD` from the section on numbers:

```
FUNC Clock$(minutes)
    DIM h = minutes \ 60
    DIM m = minutes MOD 60
    RETURN STR$(h) + ":" + RIGHT$("0" + STR$(m), 2)
ENDFUNC
PRINT Clock$(95)
PRINT Clock$(480)
PRINT Clock$(7)
```

```
1:35
8:00
0:07
```

The minutes get a leading zero by putting "0" in front and keeping the last two characters with `RIGHT$`, so seven minutes print as 0:07, the way a clock shows them. (`FORMAT$` would fill a width with spaces here, not with zeros.)

Asking a question: INPUT

A few recipes ask you something while they run. The shutdown ritual in E06 asks three questions at the end of the day, the console version of the setup wizard asks for your name and your working hours. `INPUT` shows a question and waits until you have typed an answer and pressed Enter; the answer lands in the variable named after it.

```
DIM answer$ = ""
INPUT "What did you finish today? "; answer$
PRINT "Noted: "; answer$
```

There is no output box here because the program waits for you, and the build of this book cannot type. That is also the reason the recipes keep their questions out of their modules: the module receives the answers as values, the test hands in answers of its own, and only the small program around it talks to you. When

you write a program that asks questions, this split lets you test everything except the questions themselves.

A first program of your own

To close the tour, here is a small program built from nothing, the way you would build one for a chore of your own. The chore: Jonas wants to know what the meetings of his week cost, counting the time of everyone in the room. He has a list of meetings with their length in minutes and the number of people.

The first step is to put the data into the program, as a list of maps. Later it could come from a file or from the calendar, as in E12 and E09, but while you develop it is easiest to have it right there:

```
DIM meetings = [ _
  {"title": "Team round", "minutes": 60, "people": 8}, _
  {"title": "Customer call", "minutes": 30, "people": 3}, _
  {"title": "Planning", "minutes": 90, "people": 5}]
PRINT LEN(meetings); " meetings"
PRINT meetings[2>{"title"}
```

```
3 meetings
Planning
```

The second step adds up the person minutes: the length of each meeting times the people in it. A FOR EACH loop over the list and a variable for the sum do that:

```
DIM meetings = [ _
  {"title": "Team round", "minutes": 60, "people": 8}, _
  {"title": "Customer call", "minutes": 30, "people": 3}, _
  {"title": "Planning", "minutes": 90, "people": 5}]
DIM total = 0
FOR EACH m IN meetings
  DIM cost = m{"minutes"} * m{"people"}
  PRINT FORMAT$("{<15}{>6} person minutes", m{"title"}, cost)
  total = total + cost
NEXT
PRINT FORMAT$("{<15}{>6} person minutes", "Week", total)
PRINT "That is "; ROUND(total / 60, 1); " working hours."
```

```

Team round      480 person minutes
Customer call   90 person minutes
Planning        450 person minutes
Week            1020 person minutes
That is 17 working hours.

```

The third step turns the calculation into a function, so it can be used with another week's list, and gives the result a sentence that a person understands without doing arithmetic. With an hourly rate the hours become money:

```

FUNC PersonHours(meetings)
  DIM total = 0
  FOR EACH m IN meetings
    total = total + m{"minutes"} * m{"people"}
  NEXT
  RETURN total / 60
ENDFUNC
DIM week = [{"minutes": 60, "people": 8}, _
  {"minutes": 30, "people": 3}, {"minutes": 90, "people": 5}]
DIM hours = PersonHours(week)
DIM rate = 55
PRINT FORMAT$("{:.1f} hours, about {:.0f} EUR at {} EUR an hour", _
  hours, hours * rate, rate)

```

```
17.0 hours, about 935 EUR at 55 EUR an hour
```

Three steps, each one tried before the next: data, calculation, function. That is how every recipe in this book was written, and the order is worth keeping for your own programs. A program that you build in small steps never stops working for long, and when it does, the mistake is in the few lines you just added. The meeting cost meter M12 in Chapter 4 grows from exactly this start into a program that reads your real calendar.

Switches on the command line

Most recipes can do a trial run with `--dry-run`, which shows what they would do without doing it. Such words after the program's name are called switches. `OS.ARGS()` answers everything on the command line as a list: the program's own file first, then every word you typed after it. `IN` tells whether a switch is among them:

```

DIM args = OS.ARG$()
DIM dry_run = "--dry-run" IN args
IF dry_run THEN
    PRINT "only showing what would happen"
ELSE
    PRINT "doing it"
ENDIF

```

```
doing it
```

Run as an example in this book, the program gets no switches, so it prints the second line. Typed as `jdbasic program.jdb --dry-run`, it prints the first. The shared module `WORKCONF` wraps this in `WORKCONF.FLAG("dry-run")`, and for switches that carry a value, such as `--config path\to\work.conf`, in `WORKCONF.ARG$("config")`.

Settings from work.conf

Every recipe takes its settings from one file, `work.conf`, which the setup wizard writes and which you may change in any text editor. The file is in a format called TOML: names in square brackets start a part, and every line below holds a name, an equals sign and a value. Text stands in double quotes, numbers and true or false stand alone, and lists stand in square brackets.

```

[downloads_butler]
source = "~/Downloads"
min_age_hours = 24
skip = ["desktop.ini", "thumbs.db"]

```

A recipe reads this with three lines that look the same in all of them:

```

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "downloads_butler."
DIM source$ = WORKCONF.FOLDER$(cfg, key$ + "source", "~")
DIM min_age = WORKCONF.VALUE(cfg, key$ + "min_age_hours", 24)

```

The first line finds `work.conf` and reads it. The last two take one setting each by its part and name, joined with a point, and name a value to use when the setting is missing. `FOLDER$` also turns a leading `~` into your home folder. Because every

setting has such a default, a short `work.conf` is enough, and a setting you delete by mistake does not stop the recipe.

Watch out

In TOML a backslash inside double quotes starts a special character, so `"C:\new"` does not mean what it seems. Write paths with forward slashes, `"C:/Users/Mia/new"`, or with `~`; Windows understands both.

Reading an error message

Sooner or later a program stops with a message instead of doing its work. The message says what went wrong and where. Error #13 is the kind of error, the text after it the explanation, and at line 2 the line of the program:

```
DIM files = ["offer.pdf"]  
PRINT files[3]
```

```
Error #13: Array index out of bounds: 3 at line 2
```

The list has one entry, at position 0, and the program asked for position 3. The fix is in line 2. Most messages you will meet in the recipes are of this kind and say plainly what they miss: a file that is not there, a setting with text where a number belongs, a folder without permission to write. The recipes catch the common ones with `TRY` and turn them into a sentence that tells you what to do, such as "No settings found at ... Run the setup wizard first".

When the problem is in the shape of the program itself, for example an `IF` without its `ENDIF`, `jdBasic` notices before it starts and reports a parse error. The line it names is where it stopped understanding the program, which may be some lines below the actual mistake: look upward from there for the block that was never closed.

Messages you are likely to see

These are the messages you are most likely to meet while you set up and change the recipes, with what each one means and what to do.

- **No settings found at ...:** the recipe did not find `work.conf`. Either the setup wizard has not run yet, or the recipe was started from another folder with a `--config` that points nowhere. Run the wizard, or give the path to the file after `--config`.
- **Cannot open file:** a file named in a setting does not exist. Check the spelling in `work.conf`, and check that the path uses forward slashes or `~`.
- **target exists:** `FILE.MOVE` or `FILE.COPY` refused to overwrite a file. The recipes pick a free name before they move, so this appears only when another program created the same name in the meantime. Run the recipe again.
- **is not a number:** a setting that should hold a number holds text, often because it was written in quotes. `min_age_hours = 24` is a number, `min_age_hours = "24"` is text.
- **Undeclared variable:** you have added `OPTION "EXPLICIT"` and used a name that was never created with `DIM`. Usually it is a typing mistake in the name.
- **Array index out of bounds:** a list was asked for a position it does not have. Remember that the first entry is at 0 and the last one at `LEN(list) - 1`.
- **Parse error:** the program itself is not complete, for example an `IF`, `FOR` or `FUNC` without its closing line. Look upward from the line in the message.

If a message is not in this list and the recipe's own section "When it goes wrong" does not mention it either, run the recipe with `--dry-run` first. The trial run shows how far the recipe gets before it fails, and that is usually enough to see which file or setting it was looking at.

How the recipes use all this

The table lists, for each idea of this chapter, a recipe where you can see it at work. When a listing in Chapter 3 puzzles you, the section named here is the place to look it up.

Idea	Where it appears
Lists and IN	E01 decides the kind of a file from its extension
Maps	E07 adds up hours per project
SPLIT and JOIN	E12 reads the contact list line by line

Idea	Where it appears
FOR EACH	every recipe that walks through a folder
Dates	E05, E06 and E07 work out the time of day and the weekday
TRY and CATCH	E14 survives a web page that does not answer
FUNC	every recipe, which is a module of small functions
--dry-run	every recipe that changes files
Settings	every recipe, through WORKCONF

That is the whole language as far as Chapter 3 needs it. The listings there contain nothing you have not met here, apart from functions from the modules, and each recipe explains those where it uses them.

Files and Folders

Most chores in an office are chores with files: they arrive, they pile up, they need a new name, a new place or a copy. This section shows the instructions the recipes use for that. The examples create a folder of their own with MKTEMP\$, which answers a free name in the folder for temporary files, and remove it at the end, so you can run them without worrying about your own files.

Paths

A path names a file or a folder: C:\Users\Mia\Downloads\offer.pdf. Windows separates the parts with \, other systems with /, and jdBasic accepts both. PATH.JOIN\$ puts parts together with the right separator, which is safer than joining text with +:

```
PRINT PATH.JOIN$("Reports", "2026", "march.pdf")
DIM file$ = "C:/Users/Mia/Downloads/Offer 2026.pdf"
PRINT PATH.BASENAME$(file$)
PRINT PATH.EXT$(file$)
PRINT PATH.DIRNAME$(file$)
```

```
Reports\2026\march.pdf
Offer 2026.pdf
.pdf
```

```
C:/Users/Mia/Downloads
```

PATH.EXT\$ answers the extension with its point, ".pdf". E01 removes the point and makes the rest small before it looks the extension up in its lists, so Offer.PDF and offer.pdf end up in the same folder.

In the settings file a path may start with ~, which stands for your home folder: ~/Downloads is the Downloads folder of whoever runs the recipe. The shared module WORKCONF turns that into the real path.

Reading and writing text files

TXTWRITER writes a text into a file, and TXTREADER\$ reads a whole file back as one text. With TRUE as a third value, TXTWRITER adds to the end of the file instead of replacing it; the shutdown ritual in E06 writes its journal that way.

```
DIM folder$ = MKTEMP$("book")
MKDIR folder$
DIM journal$ = PATH.JOIN$(folder$, "journal.txt")
TXTWRITER journal$, "Monday: offer sent" + CHR$(10)
TXTWRITER journal$, "Tuesday: invoices paid" + CHR$(10), TRUE
PRINT TXTREADER$(journal$)
KILL journal$
RMDIR folder$
```

```
Monday: offer sent
Tuesday: invoices paid
```

CHR\$(10) is the character that ends a line. To work through a file line by line, read it and split it there:

```
DIM folder$ = MKTEMP$("book")
MKDIR folder$
DIM todo$ = PATH.JOIN$(folder$, "todo.txt")
TXTWRITER todo$, "call Jonas" + CHR$(10) + "! send offer" + CHR$(10)
DIM lines_read = SPLIT(TRIM$(TXTREADER$(todo$)), CHR$(10))
FOR EACH task$ IN lines_read
    IF STARTSWITH(task$, "!") THEN PRINT "important: "; task$
NEXT
KILL todo$
RMDIR folder$
```

```
important: ! send offer
```

The printable week in E13 reads its to-do file exactly like this.

Asking about a file

FILE.EXISTS answers whether a file or folder is there, FILE.SIZE gives the size of a file in bytes, and FILE.STAT answers several things at once in a map: whether it exists, its size, whether it is a folder, whether it is hidden, and mtime, the date and time of its last change in your local time.

```
DIM folder$ = MKTEMP$("book")
MKDIR folder$
DIM note$ = PATH.JOIN$(folder$, "note.txt")
PRINT FILE.EXISTS(note$)
TXTWRITER note$, "hello"
PRINT FILE.EXISTS(note$), FILE.SIZE(note$)
DIM info = FILE.STAT(note$)
PRINT info{"size"}, info{"is_dir"}, info{"hidden"}
PRINT LEN(info{"mtime"}); " characters, such as 2026-10-01 18:30:00"
KILL note$
RMDIR folder$
```

```
FALSE
TRUE 5
5 FALSE FALSE
19 characters, such as 2026-10-01 18:30:00
```

The first ten characters of mtime are the date, the first seven the month. E01 sorts files into month folders with LEFT\$(mtime, 7).

Listing a folder

DIR\$ answers the names in a folder as a list. A pattern picks some of them: * stands for any characters, so *.pdf finds every PDF. The names come without the folder in front; join it back with PATH.JOIN\$ before you do anything with a file.

```
DIM folder$ = MKTEMP$("book")
MKDIR folder$
TXTWRITER PATH.JOIN$(folder$, "b.pdf"), "x"
```

```

TXTWRITER PATH.JOIN$(folder$, "a.pdf"), "x"
TXTWRITER PATH.JOIN$(folder$, "c.txt"), "x"
PRINT SORT(DIR$(PATH.JOIN$(folder$, "*")))
PRINT SORT(DIR$(PATH.JOIN$(folder$, "*.pdf")))
FOR EACH f$ IN SORT(DIR$(PATH.JOIN$(folder$, "*")))
    KILL PATH.JOIN$(folder$, f$)
NEXT
RMDIR folder$

```

```

[a.pdf, b.pdf, c.txt]
[a.pdf, b.pdf]

```

The order in which DIR\$ answers depends on the system, so the examples sort the list before they print it.

Folders: MKDIR and RMDIR

MKDIR creates a folder, together with every folder on the way that does not exist yet, and does nothing when the folder is already there. That makes it safe to call before every move. RMDIR removes an empty folder.

```

DIM folder$ = MKTEMP$("book")
DIM deep$ = PATH.JOIN$(folder$, "Sorted", "Documents", "2026-10")
MKDIR deep$
MKDIR deep$
PRINT FILE.ISDIR(deep$)
RMDIR deep$
RMDIR PATH.JOIN$(folder$, "Sorted", "Documents")
RMDIR PATH.JOIN$(folder$, "Sorted")
RMDIR folder$
PRINT FILE.EXISTS(folder$)

```

```

TRUE
FALSE

```

Copying, moving and deleting

FILE.COPY copies a file and FILE.MOVE moves or renames it. When the target is a folder that exists, the file keeps its name inside it. Both refuse to overwrite a file that is already there, unless you add TRUE as a third value, and both stop with an error that says what went wrong. KILL deletes a file.

```

DIM folder$ = MKTEMP$("book")
DIM archive$ = PATH.JOIN$(folder$, "Archive")
MKDIR archive$
DIM offer$ = PATH.JOIN$(folder$, "offer.pdf")
TXTWRITER offer$, "pdf"
FILE.COPY(offer$, PATH.JOIN$(folder$, "offer copy.pdf"))
FILE.MOVE(offer$, archive$)
PRINT SORT(DIR$(PATH.JOIN$(folder$, "*")))
PRINT DIR$(PATH.JOIN$(archive$, "*"))
TRY
    FILE.COPY(PATH.JOIN$(folder$, "offer copy.pdf"), _
        PATH.JOIN$(archive$, "offer.pdf"))
CATCH
    PRINT "refused: the archive already has offer.pdf"
ENDTRY
KILL PATH.JOIN$(archive$, "offer.pdf")
KILL PATH.JOIN$(folder$, "offer copy.pdf")
RMDIR archive$
RMDIR folder$

```

```

[Archive, offer copy.pdf]
[offer.pdf]
refused: the archive already has offer.pdf

```

Watch out

KILL does not put the file into the recycle bin; it is gone. None of the easy recipes deletes anything you created. Where a recipe removes files, such as old backups in E04, it says so in its text and only touches files it made itself.

Renaming is moving within the same folder. The batch renamer in E03 builds a new name for every file, checks that the name is free, and then calls `FILE.MOVE`:

```

DIM folder$ = MKTEMP$("book")
MKDIR folder$
DIM scan$ = PATH.JOIN$(folder$, "scan0001.pdf")
TXTWRITER scan$, "pdf"
FILE.MOVE(scan$, PATH.JOIN$(folder$, "2026-10-01 Invoice.pdf"))
PRINT DIR$(PATH.JOIN$(folder$, "*"))
KILL PATH.JOIN$(folder$, "2026-10-01 Invoice.pdf")
RMDIR folder$

```

```
[2026-10-01 Invoice.pdf]
```

Names with umlauts and other characters

File names may contain any character Windows allows: umlauts, accents, spaces, the euro sign. jdBasic keeps text as UTF-8 and passes names to Windows in that form, so a file called Rechnung Müller.pdf in your Downloads folder is found and moved like any other.

```
DIM folder$ = MKTEMP$("book")
MKDIR folder$
DIM file$ = PATH.JOIN$(folder$, "Rechnung Müller €.txt")
TXTWRITER file$, "Grüße"
PRINT DIR$(PATH.JOIN$(folder$, "*"))
PRINT TXTREADER$(file$)
KILL file$
RMDIR folder$
```

```
[Rechnung Müller €.txt]
Grüße
```

Folders inside folders

DIR\$ looks into one folder. To find every file below a folder, including the files in its subfolders and in their subfolders, a function looks into the folder, and for every subfolder it finds, calls itself on that subfolder. A function that calls itself is called *recursive*; it stops because a folder has only so many levels. The duplicate finder in E10 and the disk space report in E11 walk your folders this way.

```
FUNC AllFiles(folder$)
  DIM out = []
  FOR EACH entry$ IN SORT(DIR$(PATH.JOIN$(folder$, "*")))
    DIM full$ = PATH.JOIN$(folder$, entry$)
    IF FILE.ISDIR(full$) THEN
      out = APPEND(out, AllFiles(full$))
    ELSE
      PUSH(out, full$)
    ENDIF
  NEXT
  RETURN out
```

```
ENDFUNC
```

```
DIM root$ = MKTEMP$("book")
MKDIR PATH.JOIN$(root$, "2026", "October")
TXTWRITER PATH.JOIN$(root$, "readme.txt"), "x"
TXTWRITER PATH.JOIN$(root$, "2026", "plan.xlsx"), "x"
TXTWRITER PATH.JOIN$(root$, "2026", "October", "offer.pdf"), "x"
DIM found = AllFiles(root$)
PRINT LEN(found); " files"
FOR EACH f$ IN found
    PRINT PATH.BASENAME$(f$)
NEXT
FOR EACH f$ IN found
    KILL f$
NEXT
RMDIR PATH.JOIN$(root$, "2026", "October")
RMDIR PATH.JOIN$(root$, "2026")
RMDIR root$
```

```
3 files
offer.pdf
plan.xlsx
readme.txt
```

A walk like this can take a while in a large folder: a whole drive has hundreds of thousands of files. The recipes that walk folders start in the folder you name in `work.conf`, not at the top of the drive, and they print how far they got, so you can see that they are working.

Plan first, then act

A program that moves, renames or deletes files can do a lot of harm in a second. Every recipe in this book that changes files works in two steps, and you will see the pattern in all of them. The first step makes a *plan*: a list of what it would do, without touching anything. The second step carries the plan out. With `--dry-run`, the program stops after the first step and prints the plan.

```
FUNC Plan(names)
    DIM moves = []
    FOR EACH n$ IN names
        IF ENDSWITH(n$, ".pdf") THEN
            PUSH(moves, {"from": n$, "to": "Documents/" + n$})
        ENDIF
    NEXT
ENDFUNC
```

```

    NEXT
    RETURN moves
ENDFUNC

SUB Show(moves)
    FOR EACH m IN moves
        PRINT "would move "; m{"from"}; " -> "; m{"to"}
    NEXT
ENDSUB

DIM moves = Plan(["offer.pdf", "photo.jpg", "invoice.pdf"])
Show(moves)
PRINT LEN(moves); " files would move"

```

```

would move offer.pdf -> Documents/offer.pdf
would move invoice.pdf -> Documents/invoice.pdf
2 files would move

```

The split has three advantages. You can look at the plan before you trust the program with your files. The tests of a recipe can check the plan against what they expect, without moving a single real file. And when something does go wrong while the plan is carried out, the plan tells you exactly which files were meant to go where.

Putting it together

The last example of this chapter combines most of what you have seen: a folder with a few files, a map from extensions to kinds, a loop, a decision and a move. It is the Downloads Butler of E01 in twenty lines, without its safety checks. Read it line by line; if every line makes sense, you are ready for Chapter 3.

```

DIM folder$ = MKTEMP$("book")
MKDIR folder$
FOR EACH f$ IN ["offer.pdf", "holiday.jpg", "notes.txt"]
    TXTWRITER PATH.JOIN$(folder$, f$), "x"
NEXT
DIM kinds = {"pdf": "Documents", "txt": "Documents", "jpg": "Images"}
FOR EACH f$ IN SORT(DIR$(PATH.JOIN$(folder$, "*")))
    DIM ext$ = LCASE$(MID$(PATH.EXT$(f$), 1, 10))
    DIM kind$ = kinds{ext$} ?? "Other"
    DIM target$ = PATH.JOIN$(folder$, "Sorted", kind$)
    MKDIR target$
    FILE.MOVE(PATH.JOIN$(folder$, f$), target$)

```



```

    PRINT f$; " -> "; kind$
NEXT
FOR EACH k$, kind$ IN kinds
    DIM moved = DIR$(PATH.JOIN$(folder$, "Sorted", kind$, "*"))
    FOR EACH f$ IN moved
        KILL PATH.JOIN$(folder$, "Sorted", kind$, f$)
    NEXT
NEXT
RMDIR PATH.JOIN$(folder$, "Sorted", "Documents")
RMDIR PATH.JOIN$(folder$, "Sorted", "Images")
RMDIR PATH.JOIN$(folder$, "Sorted")
RMDIR folder$
PRINT "sorted and cleaned up"

```

```

holiday.jpg -> Images
notes.txt -> Documents
offer.pdf -> Documents
sorted and cleaned up

```

Balance dividend

This chapter gives back no minutes by itself. It is what lets you change a recipe in five minutes instead of living with a version that almost fits.

The Libraries the Recipes Use

Most office chores come down to a handful of file types and a few recurring questions. A spreadsheet has to be read or written. A Word document has to be filled. A date has to be turned into "next Monday". A mail has to be put together with an attachment. You will meet the same small set of building blocks in almost every recipe of this book, so it pays to know them before you read the recipes.

jdBasic keeps these building blocks in *libraries*: ordinary jdBasic files that someone wrote once so that nobody has to write them again. A program asks for a library with `IMPORT`. The line `IMPORT XLSX` loads the file `xlsx.jdb`, and from then on the program can call everything that file offers under the name `XLSX`, such as `XLSX.READ`. jdBasic looks for the file in a fixed order: in the folder of the program itself, in a `modules` folder next to it, in the folders listed in the environment variable `JDBASIC_PATH`, in the folder `.jdbasic\lib` inside your home

folder, and finally in the `lib` folder next to `jdbasic.exe`. The libraries of this chapter come with `jdBasic` and live in that last folder. The setup wizard puts the one module that belongs to the book, `WORKCONF`, into `.jdbasic\lib`, so every recipe finds it wherever you keep the recipe.

Do not name a program after a library

Never call your own program after a library. A program saved as `conf.jdb` that says `IMPORT CONF` finds itself first, in its own folder, and the import fails with "Undefined function". Give your programs names such as `my_settings.jdb`.

Each section below covers one library: the problem it solves, the few calls the recipes use, one example, and the recipes in Chapter 3 that rely on it. The examples are complete programs. You can paste each one into a file, run it with `jdbasic`, and see exactly the output printed under it. They work in a temporary folder made with `MKTEMP$` and clean up after themselves, so they leave nothing behind on your computer.

You do not have to learn these calls by heart. Read the sections once, then come back here when a recipe uses a library and you want to know what a line does. The full description of every library is in the `lib` folder of `jdBasic`, in a file called `<name>_lib_readme.md`.

CONF: settings you can edit by hand

Every automation needs settings: which folder to tidy, how many backups to keep, when the working day ends. They belong in a file that a person can read and change without touching the program. `CONF` reads three common formats: `dotenv` files (`KEY=value` lines), `INI` files with `[sections]`, and `TOML`, which the book uses for `work.conf`.

`TOML` looks like `INI` but knows types. Text stands in quotes, numbers and `true` or `false` stand bare, and lists stand in square brackets:

```
[backup]
folders = ["~/Documents", "~/Pictures"]
keep = 10
zip = true
```

The calls the recipes use:

- `CONF.TOML(path$)` reads a TOML file into a map. `CONF.TOML_TEXT(text$)` does the same for text you already have.
- `CONF.GET(cfg, "backup.keep", 5)` walks into the map along the dotted path and answers the value there, or the fallback 5 when the key is missing. This is the call that lets every recipe work with a short `work.conf`: whatever you leave out keeps its default.
- `CONF.ENV(path$)` and `CONF.INI(path$)` read the two other formats, for files that other programs give you.

```

IMPORT CONF

DIM L$ = CHR$(10)
DIM Q$ = CHR$(34)
DIM src$ = "[backup]" + L$ + "folders = [" + Q$ + "~/Documents" + Q$
src$ = src$ + ", " + Q$ + "~/Pictures" + Q$ + "]" + L$
src$ = src$ + "keep = 10" + L$ + "zip = true" + L$
DIM cfg = CONF.TOML_TEXT(src$)
PRINT CONF.GET(cfg, "backup.keep", 5)
PRINT JOIN(CONF.GET(cfg, "backup.folders", []), " and ")
PRINT CONF.GET(cfg, "backup.zip", FALSE)
PRINT CONF.GET(cfg, "backup.target", "~/Backups")

```

```

10
~/Documents and ~/Pictures
TRUE
~/Backups

```

The last line shows the fallback at work: the text has no target, so `CONF.GET` answers the default the program passed in.

Where the recipes use it. Every recipe reads its part of `work.conf` through the book's own module `WORKCONF`, which calls `CONF.TOML` and `CONF.GET` and adds three conveniences: it finds `work.conf` (after `--config`, in the variable `AUTOMATEWORK_CONF`, or in your work folder), it turns a leading `~` into your home folder, and it reads switches such as `--dry-run`. When you see `WORKCONF.VALUE(cfg, key$ + "keep", 10)` in a recipe, that is `CONF.GET` underneath.

Backslashes in TOML

Inside double quotes TOML reads a backslash as the start of an escape, so "C:\Users\mia" does not mean what it says. `CONF.TOML` stops at such a path with an error that names the file and the line. Write paths with forward slashes, "C:/Users/mia/Documents", or start them with `~`. Windows accepts forward slashes everywhere a program opens a file.

WORKCONF: the book's own settings module

`WORKCONF` is the one module that belongs to this book rather than to `jdBasic`. It is short, about seventy lines, and you will find it in the folder `recipes/lib` next to the recipes. It answers the questions every recipe asks before it does anything: where is `work.conf`, what does it say, and did the person start the program with `--dry-run`?

- `WORKCONF.CONFIG_PATH$()` finds the settings file. It looks after `--config` on the command line first, then in the environment variable `AUTOMATEWORK_CONF`, then in `Documents/AutomateWork/config` inside your home folder, where the setup wizard writes it.
- `WORKCONF.SETTINGS(path$)` reads the file. When it is missing, the error says so and tells you to run the wizard, instead of a recipe failing later with a puzzling message.
- `WORKCONF.VALUE(cfg, key$, fallback)` reads one value, like `CONF.GET`.
- `WORKCONF.FOLDER$(cfg, key$, fallback$)` reads a folder and turns a leading `~` into your home folder, so `~/Downloads` works on every computer.
- `WORKCONF.FLAG("dry-run")` is `TRUE` when the program was started with `--dry-run`, and `WORKCONF.ARG$("config")` answers the word after `--config`.

```
IMPORT WORKCONF
```

```
DIM folder$ = MKTEMP$("wcdemo")
MKDIR folder$
DIM file$ = folder$ + "/work.conf"
DIM L$ = CHR$(10)
DIM Q$ = CHR$(34)
DIM src$ = "[downloads_butler]" + L$ + "min_age_hours = 48" + L$
src$ = src$ + "source = " + Q$ + "~/Downloads" + Q$ + L$
```

```

TXTWRITER file$, src$
DIM cfg = WORKCONF.SETTINGS(file$)
DIM key$ = "downloads_butler."
PRINT WORKCONF.VALUE(cfg, key$ + "min_age_hours", 24)
PRINT WORKCONF.VALUE(cfg, key$ + "target", "(not set)")
DIM src_dir$ = WORKCONF.FOLDER$(cfg, key$ + "source", "")
PRINT src_dir$ = PATH.JOIN$(WORKCONF.HOME$(), "Downloads")
PRINT WORKCONF.FLAG("dry-run")
KILL file$
RMDIR folder$

```

```

48
(not set)
TRUE
FALSE

```

The third line compares instead of printing the folder, because your home folder has a different name from the one on the computer this book was built on. The last line is FALSE because the example was started without `--dry-run`.

Every recipe begins with the same three lines: `import WORKCONF` and the recipe's own module, find and read the settings, and read the values it needs with a default for each. When you write your own automations later in the book, start them the same way, and they will read the same `work.conf` as everything the wizard set up.

LOGGER: a diary for programs that run alone

A program that runs at half past six in the evening while you are already at home cannot tell you what it did. It has to write it down. **LOGGER** writes one line per event, with the time, a level and the message, to the console, to a text file, or to a file of JSON lines that other tools can read. It is called **LOGGER** because **LOG** already means the logarithm in `jdBasic`.

Levels sort the events by weight: **DEBUG** for details you only want while you look for a problem, **INFO** for the normal course of things, **WARN** for something that deserves a look, **ERROR** for a failure. A logger has a threshold, and records below it are dropped.

- `LOGGER.NEW(name$)` makes a logger; the name appears in every line.
- `LOGGER.LEVEL(lg, "INFO")` sets the threshold.

- `LOGGER.TO_FILE(lg, path$, [max_bytes], [keep])` sends the lines to a file. With `max_bytes` the file is rotated when it grows past that size, and keep old files are kept.
- `LOGGER.INFO(lg, message$, [fields])`, and `WARN`, `ERROR`, `DEBUG` in the same form, write a record. The optional map adds fields such as `files=214` to the line.

```

IMPORT LOGGER

DIM folder$ = MKTEMP$("logdemo")
MKDIR folder$
DIM file$ = folder$ + "/jobs.log"
DIM lg = LOGGER.NEW("backup")
LOGGER.LEVEL(lg, "INFO")
LOGGER.TO_FILE(lg, file$)
LOGGER.INFO(lg, "archive written", {"files": 214})
LOGGER.DEBUG(lg, "this one is below INFO")
LOGGER.WARN(lg, "disk almost full")
DIM rows = SPLIT(TRIM$(TXTREADER$(file$)), CHR$(10))
PRINT LEN(rows); " lines"
PRINT MID$(rows[0], 20, LEN(rows[0]))
PRINT MID$(rows[1], 20, LEN(rows[1]))
KILL file$
RMDIR folder$

```

```

2 lines
INFO  [backup] archive written files=214
WARN  [backup] disk almost full

```

The file has two lines, because the `DEBUG` record lies below the threshold. The example prints each line from its twentieth character on: the first twenty hold the date and the time, which differ on every run.

Where the recipes use it. The easy recipes print what they do, and the Task Scheduler keeps their output for you. The systems of Chapter 5 run for weeks, and there `LOGGER` is everywhere: the personal job server of X01 writes a log line for each run, retry and failure, and the health checks of X13 read those lines back.

DT: dates as numbers you can count with

Office work is full of dates. Invoices are due in fourteen days. Reports are for last month. A file that arrived more than a day ago should be sorted away. Dates written as text are hard to compute with, so DT turns them into plain numbers: the seconds since the first of January 1970. Two dates subtract like any two numbers, and DT does the calendar arithmetic that plain numbers cannot, such as "one month later" or "the start of the week".

- `DT.PARSE(text$)` reads the date formats that turn up in files: ISO (2026-10-30, 2026-10-30T16:45:00Z), the form mails use (Fri, 30 Oct 2026 16:45:00 +0100) and the German 30.10.2026 16:45.
- `DT.FMT$(t, "%d.%m.%Y")` writes a date in the form you want; %A is the week-day, %B the month, %H:%M the time.
- `DT.ADD(t, "+1 month -2 days")` moves a date by a spoken amount.
- `DT.DIFF(a, b, "days")` measures from one date to another in a unit.
- `DT.STARTOF(t, "week")` and `DT.ENDOF` give the first and the last moment of a day, a week (starting on Monday), a month or a year.
- `DT.IS_LEAP(year)` and `DT.DAYS_IN(year, month)` answer the calendar questions behind birthdays and month ends.

```
IMPORT DT
```

```
DIM t = DT.PARSE("2026-10-30T16:45:00Z")
PRINT DT.FMT$(t, "%A %d %B %Y, %H:%M")
PRINT DT.FMT$(DT.ADD(t, "+1 month"), "%d.%m.%Y")
DIM friday = DT.STARTOF(t, "day")
DIM first = DT.PARSE("2026-10-01")
PRINT DT.DIFF(first, friday, "days"); " days since the 1st"
PRINT DT.WEEKDAY$(t, "de")
PRINT DT.IS_LEAP(2028), DT.DAYS_IN(2026, 2)
```

```
Friday 30 October 2026, 16:45
30.11.2026
29 days since the 1st
Freitag
TRUE 28
```

DT has no time zone database. Every call takes an optional last number, the offset from UTC in hours (1 for Berlin in winter, 2 in summer), and

`DT.LOCAL_OFFSET()` tells you the offset your computer uses right now. Without the number `DT` reads and writes UTC. That is why the examples in this book write their dates with a `Z` at the end: they then print the same result on every computer.

Where the recipes use it. E09 Meeting Notes Starter finds the next meeting with `DT.PARSE` and writes its date into the notes with `DT.FMT$`. E12 Birthday Reminder counts the days to each birthday with `DT.DIFF`, and asks `DT.IS_LEAP` what to do with a birthday on the 29th of February. The core language has date functions too (`NOW`, `DATEADD`, `DATEDIFF`, `FORMAT_DATE`), which E01 uses for the age of a download; `DT` is the choice when a date comes from a file.

DF: tables, and the CSV builtins

Many chores are about tables: a list of hours, an export from the accounting program, the contacts of a team. A *data frame* is a table kept as named columns. `DF` loads one from a CSV file, adds computed columns, filters and sorts rows, groups them and sums them up, the everyday half of what a spreadsheet does, but in a program that runs the same way every week.

- `DF.READCSV(path$)` loads a CSV file with a header row; numbers come back as numbers.
- `DF.NROWS(df)` and `DF.NAMES(df)` answer the size and the column names; `DF.COL(df, "hours")` is a whole column as an array, and `DF.ROW(df, i)` one row as a map.
- `DF.WHERE(df, mask)` and `DF.SORT(df, "total", TRUE)` pick and order rows; `TRUE` sorts from the largest down.
- `DF.GROUPBY(df, "project", {"total": ["sum", "hours"]})` makes one row per project with the sum of its hours. Instead of `sum` you can ask for `mean`, `min`, `max`, `median` or `count`.
- `DF.WRITECSV(df, path$)` writes the table back.

```
IMPORT DF

DIM folder$ = MKTEMP$("dfdemo")
MKDIR folder$
DIM file$ = folder$ + "/hours.csv"
DIM rows = [{"day", "project", "hours"}, {"Mon", "Miller", 3.5}, _
            {"Mon", "Office", 2}, {"Tue", "Miller", 5}, _
```



```

["Wed", "Office", 1.5]]
CSVWRITER file$, rows
DIM hours = DF.READCSV(file$)
PRINT DF.NROWS(hours); " rows, columns "; JOIN(DF.NAMES(hours), ", ")
DIM aggs = {"total": ["sum", "hours"]}
DIM per = DF.SORT(DF.GROUPBY(hours, "project", aggs), "total", TRUE)
DIM k = 0
FOR k = 0 TO DF.NROWS(per) - 1
    DIM r = DF.ROW(per, k)
    PRINT r{"project"}; ": "; r{"total"}; " hours"
NEXT k
KILL file$
RMDIR folder$

```

```

4 rows, columns day, project, hours
Miller: 8.5 hours
Office: 3.5 hours

```

For a small file you do not always need a whole data frame. `jdBasic` has three builtins for CSV that need no `IMPORT` at all. `CSVWRITER` writes a two-dimensional array, with an optional header row; `CSVHEADER` reads the names in the first line; `CSVREADER` reads the rows back. The last argument of `CSVREADER` names the type of each column, so that a date such as 1975-11-02 stays text instead of being read as a calculation:

```

DIM folder$ = MKTEMP$("csvdemo")
MKDIR folder$
DIM file$ = folder$ + "/contacts.csv"
DIM people = [{"Ann Miller", "1980-02-29"}, _
    ["Ben Ortiz", "1975-11-02"]]
CSVWRITER file$, people, ",", ["name", "birthday"]
PRINT CSVHEADER(file$)
DIM rows = CSVREADER(file$, ",", TRUE, ["STRING", "STRING"])
PRINT LEN(rows); " contacts"
PRINT rows[1][0]; " was born on "; rows[1][1]
KILL file$
RMDIR folder$

```

```

[name, birthday]
2 contacts
Ben Ortiz was born on 1975-11-02

```

Commas inside a field

CSVWRITER puts a field that holds a comma in quotes, as the CSV format asks: "Miller, Ann". A quote inside the field is written twice. CSVREADER and DF.READCSV read such a field back as one field, and a field in quotes may even span several lines, as a spreadsheet writes a cell with a line break.

Where the recipes use it. E07 Time Tracker keeps its log with CSVWRITER and reads it back with CSVREADER. E12 Birthday Reminder reads the contact list the same way. E09 Meeting Notes Starter gets its meetings from ICAL as a data frame and reads it with DF.ROW. The report pipeline of X09 in Chapter 5 is built on DF from end to end.

XLSX: Excel files without Excel

Sooner or later every result has to go to someone who wants a spreadsheet. XLSX writes real .xlsx files that Excel, LibreOffice and Google Sheets open: several sheets, a bold header row, column widths, number formats, cell colours, formulas and frozen rows. It also reads any .xlsx back as tables of values. No Excel has to be installed, which matters for a program that runs at night on a computer where nobody is logged in.

- XLSX.NEW() starts a workbook.
- XLSX.SHEET(wb, name\$, rows) adds a sheet from a two-dimensional array; the first row becomes the bold header. A text that starts with = is a formula.
- XLSX.COLUMN(sheet, "B", {"width": 12, "format": "0.0"}) sets the width in characters and the Excel number format of a column.
- XLSX.FREEZE(sheet, 1) keeps the header row in view while you scroll.
- XLSX.WRITE(path\$, wb) writes the file. XLSX.READ(path\$) answers every sheet as a table, keyed by the sheet's name, and XLSX.SHEETS(path\$) lists the names.

```
IMPORT XLSX
```

```
DIM folder$ = MKTEMP$("xlsxdemo")
```

```
MKDIR folder$
```

```
DIM file$ = folder$ + "/week.xlsx"
```

```
DIM wb = XLSX.NEW()
```

```
DIM rows = [["Project", "Hours"], ["Miller", 8.5], ["Office", 3.5], _
```

```

["Total", "=SUM(B2:B3)"]
DIM sh = XLSX.SHEET(wb, "Week 44", rows)
XLSX.COLUMN(sh, "A", {"width": 20})
XLSX.COLUMN(sh, "B", {"format": "0.0"})
XLSX.FREEZE(sh, 1)
PRINT XLSX.WRITE(file$, wb); " parts written"
PRINT JOIN(XLSX.SHEETS(file$), ", ")
DIM back = XLSX.READ(file$)
DIM sheet = back{"Week 44"}
PRINT sheet[1][0]; " "; sheet[1][1]
PRINT LEN(sheet); " rows"
KILL file$
RMDIR folder$

```

```

7 parts written
Week 44
Miller 8.5
4 rows

```

An `.xlsx` file is a ZIP archive of XML parts; "7 parts written" is how many the workbook needed. The formula `=SUM(B2:B3)` reads back as an empty cell, because XLSX writes the formula and leaves the computing to Excel when the file is opened.

Where the recipes use it. E07 Time Tracker writes the weekly sheet of your hours. E10 Duplicate Finder writes its report of files stored twice, one row per copy. In Chapter 4, M02 Report Merger reads ten colleagues' sheets with `XLSX.READ` and writes the summary.

DOCX: Word documents without Word

Meeting notes, offers, letters and timesheets live in Word. DOCX writes `.docx` files with headings, paragraphs with **bold** and *italic* words, bullet and numbered lists, tables, page breaks, a header and a footer, pictures and a table of contents. It fills the `{{placeholders}}` of a Word document you prepared, which is how mail merge works in Chapter 4. And it reads the paragraphs and tables of any document back, so a program can check what a document says.

- `DOCX.NEW()` starts a document; `DOCX.LANGUAGE(doc, "en-US")` sets the language Word checks the spelling in.
- `DOCX.HEADING(doc, text$, [level])` and `DOCX.PARAGRAPH(doc, text$)` add text. Two stars around a word make it bold, one star italic.

- `DOCX.BULLETS(doc, items)`, `DOCX.NUMBERED(doc, items)` and `DOCX.TABLE(doc, rows)` add lists and tables.
- `DOCX.WRITE(doc, path$)` writes the file.
- `DOCX.FILL(template$, target$, values)` copies a Word file and replaces its placeholders with the values of a map.
- `DOCX.READ(path$)` answers a map with the paragraphs and the tables of a document.

```

IMPORT DOCX

DIM folder$ = MKTEMP$("docxdemo")
MKDIR folder$
DIM file$ = folder$ + "/notes.docx"
DIM doc = DOCX.NEW()
DOCX.LANGUAGE(doc, "en-US")
DOCX.HEADING(doc, "Project meeting, 30 October")
DOCX.PARAGRAPH(doc, "Present: **Mia**, Jonas, Priya")
DOCX.BULLETS(doc, ["Budget", "Next release", "Holidays"])
DOCX.TABLE(doc, [{"Task", "Who"}, {"Send the offer", "Mia"}])
DOCX.WRITE(doc, file$)
DIM back = DOCX.READ(file$)
DIM paras = FILTER(LAMBDA p$ -> p$ <> "", back{"paragraphs"})
PRINT LEN(paras); " paragraphs, the first: "; paras[0]
PRINT back{"tables"}[0][1][0]; " ("; back{"tables"}[0][1][1]; ")"
KILL file$
RMDIR folder$

```

```

5 paragraphs, the first: Project meeting, 30 October
Send the offer (Mia)

```

The heading, the paragraph and the three list items are five paragraphs. Reading the file back shows the text without its marks: the stars around *Mia* became bold type in Word, and `DOCX.READ` answers the plain words. The empty paragraphs the example filters away are the ones Word needs after a table.

Where the recipes use it. E09 Meeting Notes Starter writes the notes file for your next meeting with a heading, the attendees and an empty table for the decisions. In Chapter 4, M01 writes invoices, M03 fills letters from a template with `DOCX.FILL`, M14 compares two versions of a contract with `DOCX.READ`, and M15 writes the monthly timesheet. The build of this book also writes a Word file with `DOCX`, for reviewing.

PDFGEN: pages to print and to send

A PDF looks the same on every screen and every printer, which makes it the right format for a page you print for your desk, a card you hang in the meeting room or an invoice you send. PDFGEN writes PDF files without any other program. You place text, lines, boxes, pictures and tables on a page by position in millimetres, measured from the top left corner. Paragraphs wrap and tables continue on a new page by themselves.

- `PDFGEN.DOC({"size": "A5"})` starts a document; sizes A3, A4, A5, Letter and Legal, or any size with "width" and "height" in mm.
- `PDFGEN.ADDPAGE(pdf)` starts a page.
- `PDFGEN.USEFONT(pdf, "helvetica", "B", 16)` picks a font, a style (B bold, I italic) and a size in points. The standard fonts Helvetica, Times and Courier are always there; `PDFGEN.ADDFONT` adds a TrueType font file for any other look.
- `PDFGEN.TEXTAT(pdf, x, y, text$)` writes text at a position; `PDFGEN.MULTICELL` writes a paragraph that wraps; `PDFGEN.TABLE(pdf, rows, {"headers": [...]} )` writes a table.
- `PDFGEN.IMAGE(pdf, path$, x, y, width)` places a JPEG or PNG picture.
- `PDFGEN.WRITEFILE(pdf, path$)` writes the file; `PDFGEN.BUILD$(pdf)` answers its bytes instead.

```
IMPORT PDFGEN

DIM pdf = PDFGEN.DOC({"size": "A5"})
PDFGEN.SETINFO(pdf, "Week 44", "Mia")
PDFGEN.ADDPAGE(pdf)
PDFGEN.USEFONT(pdf, "helvetica", "B", 16)
PDFGEN.TEXTAT(pdf, 15, 20, "Week 44")
PDFGEN.USEFONT(pdf, "helvetica", "", 10)
PDFGEN.SETXY(pdf, 15, 28)
DIM todo$ = "Send the offer to Miller. Book the room for Thursday."
PDFGEN.MULTICELL(pdf, 0, 5, todo$)
DIM opts = {"headers": ["Day", "Task"]}
PDFGEN.TABLE(pdf, [{"Mon", "Offer"}, {"Thu", "Room"}], opts)
DIM bytes$ = PDFGEN.BUILD$(pdf)
PRINT LEFT$(bytes$, 8)
PRINT PDFGEN.PAGECOUNT(pdf); " page, ";
PRINT PDFGEN.PAGEWIDTH(pdf); " mm wide"
PRINT INSTR(bytes$, "(Week 44) Tj") >= 0
```

```
%PDF-1.4
1 page, 148 mm wide
TRUE
```

The example builds the PDF in memory and looks into it: every PDF file starts with %PDF, A5 is 148 millimetres wide, and the heading stands in the file as the text command (Week 44) Tj. To keep the page, replace PDFGEN.BUILD\$ with PDFGEN.WRITEFILE(pdf, "week.pdf").

Where the recipes use it. E13 Printable Week turns your to-do file into a page for the desk. E15 Guest Wi-Fi Card draws the cards with the QR code. The print and the screen edition of this book are set with PDFGEN too, with embedded TrueType fonts, bookmarks and links.

MAIL: messages built, saved and read back

Status mails, invoices, reminders: much of what an office sends is written to a pattern. MAIL builds a message the way a mail program does, with a plain text and an HTML version, pictures inside the HTML, attachments, and names and subjects with umlauts. It saves the message as an .eml file that every mail program opens, or sends it through your mail server. It also reads a saved message back into its parts, which is how a program can sort the attachments of the mails you save.

- MAIL.MESSAGE(from\$, to, subject\$) starts a message; an address may carry a name, such as Mia <mia@example.com>.
- MAIL.PLAIN(msg, text\$) and MAIL.HTMLBODY(msg, html\$) set the text and the HTML version; MAIL.ATTACH(msg, path\$) adds a file.
- MAIL.WRITEEML(msg, path\$) saves the message as a file.
- MAIL.SEND(msg, server) sends it; server is a map with the url of the mail server, the user and the password.
- MAIL.PARSEFILE(path\$) reads an .eml file back into a map with subject, from, to, text, html and the attachments.

```
IMPORT MAIL
```

```
DIM folder$ = MKTEMP$("maildemo")
MKDIR folder$
DIM file$ = folder$ + "/status.eml"
```

```

DIM msg = MAIL.MESSAGE("Mia <mia@example.com>", _
    "team@example.com", "Status, week 44")
DIM body$ = "Done: the Miller offer. Open: the room for Thursday."
MAIL.PLAIN(msg, body$)
MAIL.SETDATE(msg, "Fri, 30 Oct 2026 16:00:00 +0100")
MAIL.WRITEEML(msg, file$)
DIM back = MAIL.PARSEFILE(file$)
PRINT back{"subject"}
PRINT back{"to"}[0]
PRINT TRIM$(back{"text"})
KILL file$
RMDIR folder$

```

```

Status, week 44
team@example.com
Done: the Miller offer. Open: the room for Thursday.

```

`MAIL.SETDATE` gives the message a fixed date. Without it, a message carries the moment it was built, which is what you want in real use.

Your mail password

A program that sends mail needs the password of your mail account. The recipes of this book never store it in `work.conf`. They ask for it when they run by hand, or they put the finished message into an outbox folder as an `.eml` file that you open and send yourself.

Where the recipes use it. No easy recipe sends mail, which is why the setup wizard can promise that your files and mail stay on your computer. From Chapter 4 on, mail is everywhere: M01 sends invoices, M03 serial letters, M06 the Friday status, M07 unpacks the attachments of saved mails with `MAIL.PARSEFILE`, and X02 drafts replies to your inbox.

ICAL: calendars

Your calendar knows when you are busy, and every calendar program can save it as an `.ics` file and subscribe to one. ICAL reads and writes those files. It understands repeating events, such as "every Monday at nine, eight times", and expands them into the single dates they stand for. That turns a calendar into data a program can work with: the next meeting, the hours spent in meetings last month, the shift plan of a team.

- `ICAL.PARSEFILE(path$)` reads a calendar; `ICAL.EVENTS(cal)` answers its events.
- `ICAL.PROPS(cal, event, "SUMMARY")` reads a property of an event, such as its title, place or description.
- `ICAL.EXPAND(cal, from, to)` answers every event between two dates as a DF table with start, end and summary, repeating events expanded. `ICAL.OCCURRENCES(cal, event, from, to)` does the same for one event.
- `ICAL.NEW()`, `ICAL.ADDEVENT(cal, title$, start, end)` and `ICAL.SETPROP(cal, event, name$, value$)` build a calendar; `ICAL.WRITEFILE(cal, path$)` saves it.

```

IMPORT ICAL, DT

DIM cal = ICAL.NEW()
DIM start = DT.PARSE("2026-10-05T09:00:00Z")
DIM ev = ICAL.ADDEVENT(cal, "Team meeting", start, start + 1800)
ICAL.SETPROP(cal, ev, "RRULE", "FREQ=WEEKLY;BYDAY=M0;COUNT=8")
DIM from = DT.PARSE("2026-10-19T00:00:00Z")
DIM upto = DT.PARSE("2026-11-03T00:00:00Z")
DIM times = ICAL.OCCURRENCES(cal, ev, from, upto)
DIM k = 0
FOR k = 0 TO LEN(times) - 1
    PRINT DT.FMT$(times[k], "%a %d %b %H:%M"); " ";
    PRINT ICAL.PROPS(cal, ev, "SUMMARY")
NEXT k
PRINT LEFT$(ICAL.TEXT$(cal), 15)

```

```

Mon 19 Oct 09:00 Team meeting
Mon 26 Oct 09:00 Team meeting
Mon 02 Nov 09:00 Team meeting
BEGIN:VCALENDAR

```

The rule `FREQ=WEEKLY;BYDAY=M0;COUNT=8` is the way calendars write "weekly on Monday, eight times". `ICAL` answers the three Mondays that fall into the two weeks asked for. The last line shows the start of the text an `.ics` file holds.

Where the recipes use it. E09 Meeting Notes Starter reads your calendar file and finds the next meeting. E12 Birthday Reminder writes the birthdays of your contacts as a calendar of yearly events that you can import into Outlook or Google. In Chapter 4, M05 builds the shift and vacation plan of a team as a calendar and M12 sums up the hours you spend in meetings.

SCHED: when something should run

"Every weekday at half past seven", "on the last day of the month", "every fifteen minutes": schedules are easy to say and fiddly to compute. SCHED reads them and answers when the next run is due. It understands the five-field cron notation that servers use (`0 7 * * mon-fri`) and plain words such as `daily 18:30`, `weekly fri 16:00` and `every 90 minutes`. It also has a job loop that runs several jobs on time inside one long-running program, with retries, locks and a log.

- `SCHED.NEXTRUN(when$, t)` answers the first run after the moment `t`.
- `SCHED.UPCOMING(when$, t, n)` answers the next `n` runs.
- `SCHED.VALID(when$)` tells whether a schedule can be read, and `SCHED.PROBLEM$(when$)` why not.
- `SCHED.JOB(name$, when$, fn, [opts])` adds a job to the loop, and `SCHED.SERVE(0)` runs the loop until the program is stopped.

```
IMPORT SCHED, DT

DIM t = DT.PARSE("2026-10-30T12:00:00Z")
PRINT DT.ISO$(SCHED.NEXTRUN("daily 18:30", t))
PRINT DT.ISO$(SCHED.NEXTRUN("weekly mon 08:00", t))
DIM runs = SCHED.UPCOMING("every 90 minutes", t, 3)
PRINT JOIN(SELECT(LAMBDA r -> DT.FMT$(r, "%H:%M"), runs), " ")
PRINT SCHED.VALID("0 7 * * mon-fri"), SCHED.VALID("tomorrow")
```

```
2026-10-30T18:30:00Z
2026-11-02T08:00:00Z
13:30 15:00 16:30
TRUE FALSE
```

The 30th of October 2026 is a Friday, so the next Monday morning is the 2nd of November. An interval such as every 90 minutes counts from midnight, which is why its runs fall on 13:30, 15:00 and 16:30.

Where the recipes use it. The easy recipes leave the timing to the Windows Task Scheduler, which the setup wizard fills in for you. Their `recipe.toml` files use the same words SCHED understands, so a schedule reads the same in both places. In Chapter 5 the personal job server of X01 replaces the scattered scheduled tasks with one program built on `SCHED.JOB` and `SCHED.SERVE`.

TMPL: text with holes

A standard reply, a status mail, a web page: much of what we write is the same text with a few things changed. TMPL fills a *template*, a text with holes in double curly braces, from a map of values. Loops repeat a part for every item of a list, conditions show a part only when it applies, and filters format a value on the way in.

- `TMPL.RENDERSTR$(template$, model)` fills a template held in a text.
- `TMPL.RENDER$(path$, model)` fills a template file and keeps it in a cache until the file changes.
- In the template, `{{ name }}` is replaced by the value of `name`, `{% for i in items %}` and `{% endfor %}` repeat what stands between them, and `{% if paid %}` and `{% endif %}` show a part only when the value is there.
- Filters such as `{{ me | upper }}` or `{{ phone | default:'none' }}` change a value before it is written.

```
IMPORT TMPL

DIM L$ = CHR$(10)
DIM model AS MAP
model{"name"} = "Mr Miller"
model{"items"} = ["the offer", "the price list"]
model{"me"} = "Mia"
DIM src$ = "Dear {{ name }}, " + L$ + "attached are:" + L$
src$ = src$ + "{% for i in items %}* {{ i }}" + L$ + "{% endfor %}"
src$ = src$ + "Kind regards, {{ me | upper }}" + L$
src$ = src$ + "Phone: {{ phone | default:'see the signature' }}"
PRINT TMPL.RENDERSTR$(src$, model)
```

```
Dear Mr Miller,
attached are:
* the offer
* the price list
Kind regards, MIA
Phone: see the signature
```

The model has no phone, so the default filter writes its text instead. TMPL was made for web pages, so it escapes the characters `&` `<` `>` `"` `'` that have a meaning in HTML. For a plain mail text that is usually what you want; where it is not, three braces, `{{{ name }}}`, write a value exactly as it is.

Where the recipes use it. E08 Mail Templates keeps your standard replies as template files and fills them with the name of the person and the values from `work.conf`. In Chapter 4, M06 writes the Friday status mail from a template, and the personal dashboard of M08 renders its web page with TMPL.

Looking things up yourself

This chapter shows the calls the recipes need, not every call a library has. When you want to know more, there are three places to look, from quick to thorough.

The first is the readme of the library. Each one starts with a short example, then lists every call in a table with what it does, then explains the details that tend to surprise people, such as the offsets of DT or the formula cells of XLSX. The readmes are plain text files in the `lib` folder of `jdBasic`; any editor opens them.

The second is the library file itself. A `jdBasic` library is an ordinary program, and every function in it that you may call starts with `EXPORT` and carries a comment saying what it does. Searching the file for `EXPORT FUNC` lists them all. Reading the code of a function is the surest way to find out what it does with an unusual input.

The third is the self test. Every library has a test in the folder `tests/jdlibs` of the `jdBasic` sources, named after the library, such as `xlsx_selftest.jdb`. A test is a list of small, exact examples, each with the answer it expects: often the quickest way to see how a call behaves at its edges.

For the builtins, the functions that need no `IMPORT` such as `CSVREADER`, `FILE.MOVE` or `DIR$`, the reference is the file `help.txt` next to `jdbasic.exe`, and the command `HELP` followed by a name shows the same text in the `jdBasic` console.

Further libraries in later chapters

The medium and expert recipes reach for more of what comes with `jdBasic`. Each has its readme in the `lib` folder; the recipe that first uses one explains the calls it needs.

Library	What it does	First used in
QR	QR codes as a matrix, as text or drawn into a PDF	E15
TEXTDIFF	the differences between two texts, line by line	E14, M14

Library	What it does	First used in
REQ	HTTP requests with sessions, cookies and headers	X04, X11
JDWEB	small web applications and local web pages	M08, X03
SVG	charts and drawings as SVG pictures	M08, M09
VALID, SCHEMA	checking that data has the right form	M11
FUZZY	names that are almost the same	M11
CLI	the arguments of a command line tool	M13
RETRY	trying again after a failure, with growing pauses	X01, X11
SEARCH	full-text search over many documents	X03
JWT, OAUTH	signed tokens and logging in to web services	X04, X05
POOL	spreading work over all the cores of a computer	X08
DB	SQLite databases without hand-written SQL	X09
LLMAPI	one way to talk to a language model, local or online	X02
HTMLDOM, CACHE	reading web pages and keeping what was fetched	X11
TESTKIT, PROPTTEST	tests for your own programs	every recipe, X12
METRICS	counters and timings for programs that run alone	X13
SECRET	password hashes and random tokens	X14

TESTKIT deserves a word already here: every recipe in this book comes with a test written with it, a second program that runs the recipe against made-up files and checks each result. When you change a recipe to fit your work, run its test afterwards. If the test still passes, your change kept everything else as it was.

Running a Program on a Schedule

An automation that you have to remember to start is half an automation. Windows has a scheduler built in, the Task Scheduler, and the setup wizard uses it: for every recipe you switch on it adds one task that starts `jdbasic` with the recipe's program at the time you chose.

What the wizard puts in the Task Scheduler

Open the Start menu and type *Task Scheduler*. On the left, under *Task Scheduler Library*, you find a folder called *AutomateWork* with one task per recipe, such as *E01 Downloads Butler*. Each task runs a command like this one:

```
"C:\Users\mia\jdbasic\jdbasic.exe"  
  "C:\Users\mia\Documents\AutomateWork\recipes\  
    E01_downloads_butler\downloads_butler.jdb"  
  --config "C:\Users\mia\Documents\AutomateWork\config\work.conf"
```

It is one line in the Task Scheduler; it is broken here to fit the page. The `--config` part tells the recipe which settings file to read, so a recipe started by the scheduler finds exactly the settings the wizard wrote.

You can look at the same list from a command prompt:

```
schtasks /Query /TN "AutomateWork\E01 Downloads Butler"
```

and start a task right away, which is the quickest way to see whether it works:

```
schtasks /Run /TN "AutomateWork\E01 Downloads Butler"
```

Watch out

A scheduled task runs only while you are logged in, and a laptop that sleeps at 18:30 misses the Downloads Butler that evening. The tasks the wizard creates do not catch up later; for the easy recipes that does no harm, since the next run does the work of both. For anything that must never be missed, Chapter 5 has a job server that catches up on its own.

Changing a schedule

Run the wizard again, pick the recipe on the third page, change the schedule and set up. The wizard replaces the old task with the new one. To stop a recipe, untick *Use this recipe* and set up again, or remove all tasks at once with the console wizard:

```
jdbasic wizard\setup_console.jdb --uninstall
```

Your work folder and your settings stay where they are, so switching a recipe back on later costs one click.

When a schedule is not enough

The Task Scheduler starts a program at a time. Some jobs need more: run every five minutes but never twice at the same time, try again after a failure, write down what happened. The SCHED library does that inside jdBasic, and Chapter 5 builds a personal job server on it that runs all your automations from one place. For the easy recipes, the Task Scheduler is all you need.

When Something Goes Wrong

Programs fail. A folder was renamed, a file is open in another program, a setting has a typing mistake. jdBasic tells you what happened and where; the trick is to read the message calmly.

Reading an error message

Here is a small program with a mistake. The list has two entries, at the positions 0 and 1, and the program asks for position 2:

```
DIM files = ["a.pdf", "b.pdf"]  
PRINT files[2]
```

```
Error #13: Array index out of bounds: 2 at line 2
```

The message has three parts. *Error #13* is the kind of error; the number helps when you search the documentation. *Array index out of bounds: 2* says what went wrong in plain words: there is no entry 2. *at line 2* says where. Open the file, go to that line, and the mistake is usually right there or one line above.

Checking a program without running it

`jdbasic --lint file.jdb` reads a program and reports what it can find without running a single line. That is useful before you schedule a changed recipe:

```
jdbasic --lint downloads_butler.jdb
```

When everything is fine, it says `LINT: Parsed OK.` and lists what it found. When a block is not closed, for example an `IF` without its `ENDIF`, it reports the line where it reached the end of the file still waiting, and words it as an `END` it expected:

```
LINT error: Parse error at line 4: expected 'END', got ''
```

The line number points to the end of the program, not to the `IF`. Look upwards for the block that is missing its end.

Dry runs

Every recipe of this book that changes files has a switch `--dry-run`. With it, the recipe prints what it would do and does nothing. Use it after every change of a setting:

```
jdbasic downloads_butler.jdb --dry-run
```

It is the same habit the recipes follow on the inside: first make a plan, then show it, then carry it out.

The doctor

When a scheduled recipe stops working and you do not know why, ask the wizard:

```
jdbasic wizard\setup_console.jdb --doctor
```

It checks that the settings file exists, that `jdbasic` is where the settings say, that every recipe you switched on is in the work folder, and that its task is in the Task Scheduler. Every problem it finds is one line, such as:

```
E01: no scheduled task AutomateWork\E01 Downloads Butler
```

Running the wizard again repairs most of what the doctor finds.

Try this

Keep a text file `problems.txt` in your work folder. Every time a recipe fails, write down the date, the message and what fixed it. After a few months it is the most useful page of this book.

From Script to Program

A `.jdb` file needs `jdbasic` to run. `jdBasic` can also turn a program into an `.exe` of its own, which runs on another computer without `jdBasic` installed:

```
jdbasic -c downloads_butler.jdb
```

This writes `downloads_butler.exe` and the libraries it needs next to it. Copy the folder to a colleague's computer and the program runs there with a double click. The compiled program is also faster, which matters for the large jobs of Chapter 5.

Compiling needs the Microsoft C++ build tools on the computer that compiles, not on the one that runs the result. The forms release this book starts with brings the compiler with it; `jdbasic --version` lists *NativeC* among its features. Every recipe in this book was compiled and tested as an `.exe` as well, so nothing changes when you compile one.

Watch out

A program you hand to a colleague runs with that colleague's rights and on that colleague's files. Recipe X07 in Chapter 5 shows how to give a tool settings of its own, a version number and an easy way to update it, so that you do not become everybody's help desk.

Chapter 3: Easy: Fifteen Automations You Switch On

The fifteen recipes in this chapter need no programming. The setup wizard from Chapter 2 copies them to your work folder, asks you a few questions and plans when each one runs. What you can change lives in one file, `work.conf`, and every recipe tells you which lines in it belong to it.

You will still see the programs. Each recipe prints the program that does the work, and a short walk through it. Read it once: a program you have read is a program you trust, and the day you want it to do one thing differently you will know where to look.

Each recipe follows the same order:

1. **The chore:** what it costs you now.
2. **What you get:** the result in one picture or one sentence.
3. **Before you start:** what has to be in place.
4. **The program:** the code, exactly as it was tested.
5. **How it works:** the program step by step.
6. **Run it:** the command and what it prints.
7. **Schedule it:** how it runs without you.
8. **Make it yours:** the settings you can change.
9. **When it goes wrong:** the problems you are likely to meet.
10. **Balance dividend:** the time it gives back.

After the fifteen come two bonus recipes. E16 Form Filler types into a form for you, which means it takes over your keyboard for a few seconds, so the wizard leaves it switched off until you choose it. E17 Break Arcade is a game of 2048 for

the length of a break. E18 Office to PDF turns a folder of Word and PowerPoint files into PDFs through Office itself; it needs Office installed, so it also starts switched off.

E01 Downloads Butler

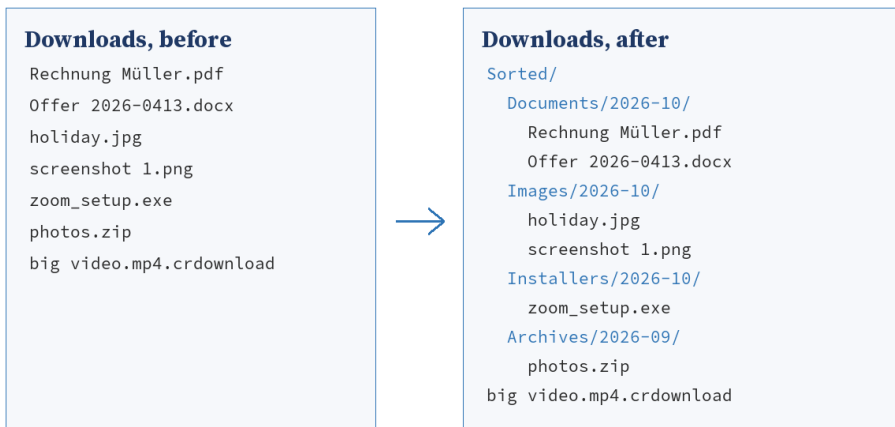
The chore

Mia is a project assistant. Every file anyone sends her ends up in the Downloads folder: invoices, offers, photos from the last site visit, the installer for the video call tool, the zip file with last year's figures. On Monday morning the folder holds three hundred files. Finding the offer for the Miller project means scrolling, sorting by date, guessing at names, and opening two wrong PDFs first.

Once a month Mia spends half an hour tidying it up. By the middle of the next week it looks the same as before.

What you get

Every night the Downloads Butler looks at the Downloads folder. Each file that has rested there for a day moves into a folder for its kind and the month it arrived: documents to Sorted/Documents/2026-10, photos to Sorted/Images/2026-10, and so on. Downloads that are still running stay where they are, and so does anything that arrived today, because that is the file you are most likely to need next.



The Downloads folder before and after the Downloads Butler ran

Before you start

The setup wizard has run and the Downloads Butler is switched on. That is all; the wizard found your Downloads folder and wrote it into `work.conf`.

The program

The recipe has two files. The first is the program the wizard runs. It reads the settings, asks the butler for a plan and carries it out:

```
' Downloads Butler: sorts the downloads folder into one folder per
' kind of file and month. The settings are the [downloads_butler]
' part of work.conf.
'
' jdbasic downloads_butler.jdb          sort now
' jdbasic downloads_butler.jdb --dry-run  show what would move

IMPORT WORKCONF, BUTLER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "downloads_butler."
DIM source$ = WORKCONF.FOLDER$(cfg, key$ + "source", "~/Downloads")
DIM sorted$ = PATH.JOIN$(source$, "Sorted")
DIM target$ = WORKCONF.FOLDER$(cfg, key$ + "target", sorted$)
DIM min_age = WORKCONF.VALUE(cfg, key$ + "min_age_hours", 24)
DIM defaults = BUTLER.DEFAULT_RULES()
DIM rules = WORKCONF.VALUE(cfg, key$ + "rules", defaults)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM moves = BUTLER.PLAN(source$, target$, rules, min_age, NOW())
DIM n_sorted = BUTLER.APPLY(moves, dry_run)
IF n_sorted = 0 THEN
    PRINT "Nothing to sort in "; source$
ELSEIF dry_run THEN
    PRINT n_sorted; " files would move. Run without --dry-run."
ELSE
    PRINT n_sorted; " files sorted."
ENDIF
```

The second file is the butler itself. It knows which extension belongs to which kind of file, which files to leave alone, and how to find a free name when a file of the same name is already in the target folder:

```

' butler.jdb: sorts a downloads folder. Every file that has rested
' long enough moves into a folder for its kind and the month it
' arrived, such as Sorted/Documents/2026-10/.
,
'   IMPORT BUTLER
'   DIM rules = BUTLER.DEFAULT_RULES()
'   DIM moves = BUTLER.PLAN(src$, dst$, rules, 24, NOW())
'   PRINT BUTLER.APPLY(moves, FALSE); " files sorted"

EXPORT MODULE BUTLER

' The kinds of files and their extensions.
EXPORT FUNC DEFAULT_RULES()
  DIM rules AS MAP
  rules{"Documents"} = ["pdf", "doc", "docx", "odt", "rtf", _
    "txt", "md", "xls", "xlsx", "ods", "csv", "ppt", "pptx", _
    "odp"]
  rules{"Images"} = ["jpg", "jpeg", "png", "gif", "webp", "heic", _
    "bmp", "svg", "tif", "tiff"]
  rules{"Audio and Video"} = ["mp3", "wav", "m4a", "flac", "mp4", _
    "mov", "mkv", "avi", "webm"]
  rules{"Archives"} = ["zip", "7z", "rar", "tar", "gz", "tgz"]
  rules{"Installers"} = ["exe", "msi", "msix", "dmg", "pkg", _
    "deb", "rpm"]
  RETURN rules
ENDFUNC

' The kind a file belongs to by its extension, "Other" when no rule
' names the extension.
EXPORT FUNC KIND$(file_name$, rules)
  DIM ext$ = LCASE$(PATH.EXT$(file_name$))
  IF ext$ = "" THEN RETURN "Other"
  ext$ = MID$(ext$, 1, LEN(ext$))
  DIM kinds = MAP.KEYS(rules)
  DIM k = 0
  FOR k = 0 TO LEN(kinds) - 1
    DIM kind$ = kinds[k]
    IF ext$ IN rules{kind$} THEN RETURN kind$
  NEXT k
  RETURN "Other"
ENDFUNC

' TRUE for a partial download (a browser's temporary file).
FUNC Unfinished(file_name$)
  DIM ext$ = LCASE$(PATH.EXT$(file_name$))
  DIM partial = [".crdownload", ".part", ".partial", ".download", _
    ".tmp"]
  RETURN ext$ IN partial

```

```

ENDFUNC

' A free file name in a folder: the name itself, or "name (2).ext",
' "name (3).ext" and so on. Names in taken count as used.
EXPORT FUNC FREE_NAME$(folder$, file_name$, taken)
    DIM candidate$ = PATH.JOIN$(folder$, file_name$)
    DIM used = FILE.EXISTS(candidate$) ORELSE candidate$ IN taken
    IF NOT used THEN RETURN candidate$
    DIM ext$ = PATH.EXT$(file_name$)
    DIM stem$ = LEFT$(file_name$, LEN(file_name$) - LEN(ext$))
    DIM n = 2
    DO
        DIM numbered$ = stem$ + " (" + STR$(n) + ")" + ext$
        candidate$ = PATH.JOIN$(folder$, numbered$)
        n = n + 1
    LOOP WHILE FILE.EXISTS(candidate$) ORELSE candidate$ IN taken
    RETURN candidate$
ENDFUNC

' TRUE for an entry the butler leaves alone: a folder, a hidden file
' or a partial download.
FUNC LeaveAlone(file_name$, info)
    IF info{"is_dir"} ORELSE info{"hidden"} THEN RETURN TRUE
    RETURN Unfinished(file_name$)
ENDFUNC

' The moves that sort the source folder: a map per file with "from",
' "to" and "short" ("to" inside the target). Files younger than
' min_age_hours stay.
EXPORT FUNC PLAN(source$, target$, rules, min_age_hours, now)
    DIM moves = []
    DIM taken = []
    DIM entries = DIR$(PATH.JOIN$(source$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(entries) - 1
        DIM file_name$ = entries[k]
        DIM from$ = PATH.JOIN$(source$, file_name$)
        DIM info = FILE.STAT(from$)
        IF NOT LeaveAlone(file_name$, info) THEN
            DIM changed$ = info{"mtime"}
            DIM age = DATEDIFF("H", CDATE(changed$), now)
            IF age >= min_age_hours THEN
                DIM kind$ = KIND$(file_name$, rules)
                DIM month$ = LEFT$(changed$, 7)
                DIM folder$ = PATH.JOIN$(target$, kind$, month$)
                DIM to$ = FREE_NAME$(folder$, file_name$, taken)
                PUSH(taken, to$)
                DIM new$ = PATH.BASENAME$(to$)
            
```

```

        DIM short$ = PATH.JOIN$(kind$, month$, new$)
        PUSH(moves, {"from": from$, "to": to$, _
                    "short": short$})
    ENDIF
ENDIF
NEXT k
RETURN moves
ENDFUNC

' Carries out the moves and prints a line for each; with dry_run it
' only prints them. Answers the number of files moved, or that would.
EXPORT FUNC APPLY(moves, dry_run)
    DIM k = 0
    FOR k = 0 TO LEN(moves) - 1
        DIM m = moves[k]
        DIM from$ = m{"from"}
        DIM to$ = m{"to"}
        DIM file$ = PATH.BASENAME$(from$)
        DIM short$ = m{"short"}
        IF dry_run THEN
            PRINT "would move "; file$; " -> "; short$
        ELSE
            MKDIR PATH.DIRNAME$(to$)
            FILE.MOVE(from$, to$)
            PRINT "moved "; file$; " -> "; short$
        ENDIF
    NEXT k
    RETURN LEN(moves)
ENDFUNC

```

How it works

1. WORKCONF.SETTINGS reads work.conf. The program takes four values from it: the folder to tidy, the folder to sort into, how old a file must be, and the list of kinds. Each has a default, so a short work.conf works too.
2. BUTLER.PLAN looks at every entry of the Downloads folder with DIR\$. It skips folders, hidden files and partial downloads such as .crdownload, and every file younger than min_age_hours.
3. For each file that stays in the plan, KIND\$ finds its kind from the extension, and the first seven characters of its change date give the month, such as 2026-10.
4. FREE_NAME\$ makes sure nothing is overwritten. If holiday.jpg is already in the target folder, the new one becomes holiday (2).jpg.

5. BUTLER.APPLY creates the folders with MKDIR and moves each file with FILE.MOVE. With `--dry-run` it only prints what it would do.

Planning first and moving second is a habit worth copying for every program that changes files: the plan can be shown, checked and tested before anything happens.

Run it

Try it with `--dry-run` first. Nothing moves; you see the plan:

```
jdbasic downloads_butler.jdb --dry-run
would move holiday.jpg -> Images\2026-10\holiday.jpg
would move budget.xlsx -> Documents\2026-10\budget.xlsx
2 files would move. Run without --dry-run.
```

When the plan looks right, run it without the switch.

Schedule it

The wizard plans the Downloads Butler for every evening at 18:30. To pick another time, run the wizard again and change it on the page "Schedule".

Make it yours

The butler's settings are in the part of `work.conf` that starts with `[downloads_butler]:`

```
[downloads_butler]
source = "~/Downloads"
target = "~/Downloads/Sorted"
min_age_hours = 24

[downloads_butler.rules]
Documents = ["pdf", "docx", "xlsx", "pptx", "txt", "csv"]
Images = ["jpg", "jpeg", "png", "heic"]
"Audio and Video" = ["mp3", "mp4", "mov"]
Archives = ["zip", "7z"]
Installers = ["exe", "msi"]
```

- **Sort into your Documents folder** instead of a folder inside Downloads: set `target = "~/Documents/From Downloads"`.

- **Keep files longer:** `min_age_hours = 72` leaves everything of the last three days where it is.
- **Add a kind:** a line such as `CAD = ["dwg", "dxf", "step"]` under `[downloads_butler.rules]` gives drawings their own folder. Files with an extension no rule names go to `Other`.

When it goes wrong

- **"No settings found":** the program did not find `work.conf`. Run the wizard, or give the file with `--config path\to\work.conf`.
- **A file stays where it is:** it is younger than `min_age_hours`, it is hidden, or its name ends in `.crdownload` or `.part`, which browsers use while a download is running.
- **"target exists":** two files with the same name arrived in the same run and something created the second one in between. Run the butler again; it picks the next free name.

Balance dividend

About 15 minutes a week of tidying and searching, and the half hour once a month that the folder used to cost you.

E02 Morning Launcher

The chore

Every morning Mia opens the same things in the same order: the mail in the browser, the project folder, the planning sheet, the team chat. On Mondays the planning tool comes first, on Fridays the week report. Twenty clicks before the first real piece of work, and on a busy morning she forgets one and notices it only when somebody asks.

What you get

When the workday starts, the Morning Launcher opens everything on your list: programs, folders, files and web pages. Each weekday can add its own things to the list you open every day.

Before you start

The setup wizard has run and the Morning Launcher is switched on. The wizard asks you for the first few things to open; you can add more later in `work.conf`.

The program

The program reads your lists, asks the launcher for today's plan and opens it:

```
' Morning Launcher: opens the programs, folders and web pages of the
' workday. The lists are the [morning_launcher] part of work.conf.
'
'   jdbasic morning_launcher.jdb          open everything
'   jdbasic morning_launcher.jdb --dry-run  show what would open

IMPORT WORKCONF, LAUNCHER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM empty AS MAP
DIM settings = WORKCONF.VALUE(cfg, "morning_launcher", empty)
DIM dry_run = WORKCONF.FLAG("dry-run")
DIM today = CINT(FORMAT_DATE(NOW(), "%w"))

DIM plan = LAUNCHER.PLAN(settings, today, OS.GETOS())
DIM n_open = LAUNCHER.START(plan, dry_run)
IF n_open = 0 THEN
```

```

    PRINT "Nothing to open on "; LAUNCHER.DAY_KEY$(today); "."
ELSEIF dry_run THEN
    PRINT n_open; " things would open. Run without --dry-run."
ELSE
    PRINT "Good morning. "; n_open; " things are opening."
ENDIF

```

The launcher builds the plan. It knows the weekdays and which program opens a target on your system:

```

' launcher.jdb: the plan for the start of a workday. The things to
' open every day, plus those of one weekday, each as the program and
' the argument that opens it.
'
'   IMPORT LAUNCHER
'   DIM plan = LAUNCHER.PLAN(settings, 1, "WINDOWS")
'   PRINT LAUNCHER.START(plan, TRUE); " things would open"

EXPORT MODULE LAUNCHER

' The settings key of a weekday, 0 for Sunday to 6 for Saturday.
EXPORT FUNC DAY_KEY$(weekday)
    DIM keys = ["sun", "mon", "tue", "wed", "thu", "fri", "sat"]
    RETURN keys[weekday MOD 7]
ENDFUNC

' What a target is: "web" for an address, "app" for a program file,
' "path" for any other file or folder.
EXPORT FUNC KIND$(target$)
    DIM low$ = LCASE$(target$)
    IF STARTSWITH(low$, "http://") THEN RETURN "web"
    IF STARTSWITH(low$, "https://") THEN RETURN "web"
    IF ENDSWITH(low$, ".exe") ORELSE ENDSWITH(low$, ".lnk") THEN
        RETURN "app"
    ENDIF
    RETURN "path"
ENDFUNC

' The program that opens a target on a system: Explorer on Windows,
' open on macOS, xdg-open elsewhere.
EXPORT FUNC OPENER$(os$)
    IF os$ = "WINDOWS" THEN RETURN "explorer"
    IF os$ = "MACOS" THEN RETURN "open"
    RETURN "xdg-open"
ENDFUNC

' A list setting as an array of texts; a missing key is an empty list.

```

```

FUNC ListOf(settings, key$)
    DIM out = []
    IF NOT MAP.EXISTS(settings, key$) THEN RETURN out
    DIM items = settings[key$]
    DIM k = 0
    FOR k = 0 TO LEN(items) - 1
        DIM item$ = items[k]
        PUSH(out, item$)
    NEXT k
    RETURN out
ENDFUNC

' The launches for a weekday: the "every_day" list first, then the
' list of that day, each target once. A launch is a map with
' "target", "kind", "program" and "args".
EXPORT FUNC PLAN(settings, weekday, os$)
    DIM wanted = APPEND(ListOf(settings, "every_day"), _
        ListOf(settings, DAY_KEY$(weekday)))
    DIM seen AS MAP
    DIM plan = []
    DIM k = 0
    FOR k = 0 TO LEN(wanted) - 1
        DIM target$ = wanted[k]
        DIM fresh = NOT MAP.EXISTS(seen, target$)
        IF TRIM$(target$) <> "" AND ALSO fresh THEN
            seen{target$} = TRUE
            DIM launch AS MAP
            launch{"target"} = target$
            launch{"kind"} = KIND$(target$)
            launch{"program"} = OPENER$(os$)
            launch{"args"} = [target$]
            PUSH(plan, launch)
        ENDIF
    NEXT k
    RETURN plan
ENDFUNC

' Opens every target of the plan and prints a line for each; with
' dry_run it only prints them. Answers the number of targets.
EXPORT FUNC START(plan, dry_run)
    DIM k = 0
    FOR k = 0 TO LEN(plan) - 1
        DIM launch = plan[k]
        DIM target$ = launch{"target"}
        IF dry_run THEN
            PRINT "would open "; target$
        ELSE
            DIM r = OS.EXEC(launch{"program"}, launch{"args"})

```

```

        PRINT "opened "; target$
    ENDIF
NEXT k
RETURN LEN(plan)
ENDFUNC

```

How it works

1. WORKCONF.VALUE reads the [morning_launcher] part of work.conf as one map of lists.
2. FORMAT_DATE(NOW(), "%w") gives the weekday as a number, 0 for Sunday, and DAY_KEY\$ turns it into the key of that day's list, such as mon.
3. PLAN puts the every_day list first and the day's own list after it, and opens each target only once, even when both lists name it.
4. Every target is opened by the program that opens things on your system: Explorer on Windows. Explorer starts a program, shows a folder, opens a file with the program it belongs to and a web address in your browser.
5. START calls OS.EXEC with the opener and the target as an argument list. A target is never joined into a command line, so a folder called R&D opens as a folder and runs nothing else.

Run it

```

jdbasic morning_launcher.jdb --dry-run
would open https://mail.example.com
would open C:\Work\Projects
would open C:\Tools\planner.exe
3 things would open. Run without --dry-run.

```

Schedule it

The wizard plans the Morning Launcher for every workday at 08:00, or for the moment you log in, whichever you pick on the page "Schedule".

Make it yours

```
[morning_launcher]
every_day = ["https://mail.example.com", "~/Documents/Projects"]
mon = ["C:/Tools/planner.exe"]
fri = ["~/Documents/Reports/week.xlsx"]
```

- **Add a day:** the keys are mon, tue, wed, thu, fri, sat and sun.
- **Open a file:** name it with its path; it opens in the program it belongs to.
- **Open a program:** name its .exe file or a shortcut (.lnk) from the start menu.

When it goes wrong

- **Nothing opens:** run with `--dry-run`. An empty plan means the lists in `work.conf` are empty for today, or the section is misspelled.
- **A program does not start:** open its path in Explorer by hand. If that fails too, the path is wrong; copy it from the program's shortcut.
- **Everything opens twice:** the scheduler and your login both start the launcher. Keep one of the two in the wizard.

Balance dividend

About 10 minutes a week, and a calmer first quarter of an hour.

E03 Batch Renamer

The chore

The office scanner names every page scan 4711.pdf. The phone names a photo IMG_20261001_0932.jpg. Mia renames them by hand so she can find them again: the date first, then what it is. Twenty files a week, each with a right click, a rename and a typo now and then.

What you get

The Batch Renamer puts the day a file arrived in front of its name: scan 4711.pdf becomes 2026-10-01 scan 4711.pdf. Sorted by name, the folder is sorted by date, and the name still tells you what it is. Files that already start with a date stay as they are.

Before you start

The setup wizard has run and the Batch Renamer is switched on. It asked you which folder to tidy; the scanner's folder is a good first choice.

The program

The program reads the settings, plans the new names and applies them:

```
' Batch Renamer: gives the files of a folder names that start with
' the day they arrived, such as "2026-10-01 scan 4711.pdf". The
' settings are the [batch_renamer] part of work.conf.
'
' jdbasic batch_renamer.jdb          rename now
' jdbasic batch_renamer.jdb --dry-run show the new names

IMPORT WORKCONF, RENAMER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "batch_renamer."
DIM scans$ = "~/Documents/Scans"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", scans$)
DIM usual$ = "{date} {name}{ext}"
DIM pattern$ = WORKCONF.VALUE(cfg, key$ + "pattern", usual$)
DIM extensions = WORKCONF.VALUE(cfg, key$ + "extensions", [])
DIM dry_run = WORKCONF.FLAG("dry-run")
```

```

DIM plan = RENAMER.PLAN(folder$, pattern$, extensions)
DIM n_renamed = RENAMER.APPLY(plan, dry_run)
IF n_renamed = 0 THEN
    PRINT "Nothing to rename in "; folder$
ELSEIF dry_run THEN
    PRINT n_renamed; " files would get a new name."
ELSE
    PRINT n_renamed; " files renamed."
ENDIF

```

The renamer makes the names and checks that none is taken:

```

' renamer.jdb: new names for the files of a folder from a pattern.
' {date} is the day the file last changed, {name} its name without
' the extension, {ext} the extension with its dot.
'
'   IMPORT RENAMER
'   DIM plan = RENAMER.PLAN("C:/Scans", "{date} {name}{ext}", ["pdf"])
'   PRINT RENAMER.APPLY(plan, TRUE); " files would get a new name"

EXPORT MODULE RENAMER

' A file name that starts with a date such as 2026-10-01.
EXPORT FUNC DATED(file_name$)
    RETURN REGEX.MATCH("^\\d{4}-\\d{2}-\\d{2}.*", file_name$)
ENDFUNC

' The name the pattern gives a file that changed on date$.
EXPORT FUNC NEW_NAME$(pattern$, file_name$, date$)
    DIM ext$ = PATH.EXT$(file_name$)
    DIM stem$ = LEFT$(file_name$, LEN(file_name$) - LEN(ext$))
    DIM out$ = REPLACE$(pattern$, "{date}", date$)
    out$ = REPLACE$(out$, "{name}", stem$)
    RETURN TRIM$(REPLACE$(out$, "{ext}", ext$))
ENDFUNC

' A free file name in a folder: the name itself, or "name (2).ext",
' "name (3).ext" and so on. Names in taken count as used.
EXPORT FUNC FREE_NAME$(folder$, file_name$, taken)
    DIM candidate$ = PATH.JOIN$(folder$, file_name$)
    DIM used = FILE.EXISTS(candidate$) ORELSE candidate$ IN taken
    IF NOT used THEN RETURN candidate$
    DIM ext$ = PATH.EXT$(file_name$)
    DIM stem$ = LEFT$(file_name$, LEN(file_name$) - LEN(ext$))
    DIM n = 2
    DO

```



```

        DIM numbered$ = stem$ + " (" + STR$(n) + ")" + ext$
        candidate$ = PATH.JOIN$(folder$, numbered$)
        n = n + 1
    LOOP WHILE FILE.EXISTS(candidate$) ORELSE candidate$ IN taken
    RETURN candidate$
ENDFUNC

' TRUE when a file's extension is in the list; an empty list takes
' every file.
FUNC Wanted(file_name$, extensions)
    IF LEN(extensions) = 0 THEN RETURN TRUE
    DIM ext$ = LCASE$(PATH.EXT$(file_name$))
    IF ext$ = "" THEN RETURN FALSE
    RETURN MID$(ext$, 1, LEN(ext$)) IN extensions
ENDFUNC

' The renames for a folder: a map per file with "from", "to" and
' "new". Folders, hidden files, files outside the extensions and
' files whose name already starts with a date are left alone.
EXPORT FUNC PLAN(folder$, pattern$, extensions)
    DIM plan = []
    DIM taken = []
    DIM entries = SORT(DIR$(PATH.JOIN$(folder$, "*")))
    DIM k = 0
    FOR k = 0 TO LEN(entries) - 1
        DIM file_name$ = entries[k]
        DIM from$ = PATH.JOIN$(folder$, file_name$)
        DIM info = FILE.STAT(from$)
        DIM skip = info{"is_dir"} ORELSE info{"hidden"}
        IF NOT skip AND ALSO Wanted(file_name$, extensions) THEN
            IF NOT DATED(file_name$) THEN
                DIM day$ = LEFT$(info{"mtime"}, 10)
                DIM new$ = NEW_NAME$(pattern$, file_name$, day$)
                DIM to$ = FREE_NAME$(folder$, new$, taken)
                PUSH(taken, to$)
                DIM shown$ = PATH.BASENAME$(to$)
                PUSH(plan, {"from": from$, "to": to$, "new": shown$})
            ENDIF
        ENDIF
    NEXT k
    RETURN plan
ENDFUNC

' Renames the files and prints a line for each; with dry_run it only
' prints them. Answers the number of files in the plan.
EXPORT FUNC APPLY(plan, dry_run)
    DIM k = 0
    FOR k = 0 TO LEN(plan) - 1

```

```

    DIM r = plan[k]
    DIM old$ = PATH.BASENAME$(r{"from"})
    DIM new$ = r{"new"}
    IF dry_run THEN
        PRINT "would rename "; old$; " -> "; new$
    ELSE
        FILE.MOVE(r{"from"}, r{"to"})
        PRINT "renamed "; old$; " -> "; new$
    ENDIF
NEXT k
RETURN LEN(plan)
ENDFUNC

```

How it works

1. The pattern is text with three marks: {date} for the day the file last changed, {name} for its name without the extension, {ext} for the extension with its dot.
2. PLAN looks at every file of the folder. It leaves out folders, hidden files, files whose extension is not in your list and files whose name already starts with a date, so a second run changes nothing.
3. NEW_NAME\$ fills the pattern. FREE_NAME\$ adds (2), (3) and so on when the new name is taken, by a file in the folder or by an earlier file of the same plan.
4. APPLY renames each file with FILE.MOVE, or only prints the plan with --dry-run.

Run it

```

jdbasic batch_renamer.jdb --dry-run
would rename scan 4711.pdf -> 2026-10-01 scan 4711.pdf
would rename IMG_0932.jpg -> 2026-10-01 IMG_0932.jpg
2 files would get a new name.

```

Schedule it

The wizard plans the Batch Renamer for every evening at 18:00. Files you scan during the day carry their date the next morning.

Make it yours

```
[batch_renamer]
folder = "~/Documents/Scans"
pattern = "{date} {name}{ext}"
extensions = ["pdf", "jpg", "png"]
```

- **Date last:** pattern = "{name} {date}{ext}". Files that end with a date are not recognised as done, so use this only for a folder you rename once.
- **Every kind of file:** extensions = [].
- **Another folder:** run the wizard again or change folder.

When it goes wrong

- **A file gets the wrong date:** the date is when the file last changed. A file copied from an old disk may carry the day it was copied.
- **A name ends in (2):** a file of that name was already there. Nothing was overwritten; compare the two and delete one.
- **Nothing happens:** the files already start with a date, or their extension is not in the list.

Balance dividend

About 10 minutes a week, and every scan found by its date.

E04 One-Click Backup

The chore

Mia's project folders live on her laptop. The company has a backup for the file server, not for her desktop. Once in a while she copies the important folders to a USB stick, when she remembers, and she does not remember which of the five copies on the stick is the newest.

What you get

Every Friday afternoon the One-Click Backup packs the folders you name into one ZIP archive with the date in its name, such as backup 2026-10-02 1630.zip. It keeps the newest ten archives and deletes older ones, so the backup drive does not fill up. Every ZIP opens with a double click in Explorer.

Before you start

The setup wizard has run and the One-Click Backup is switched on. Plug in the drive you want to keep the archives on and name it as the target in the wizard. A folder on the same disk protects you from mistakes; only another drive protects you from a broken laptop.

The program

The program reads the settings, packs the archive and cleans up:

```
' One-Click Backup: packs the folders you name into one dated ZIP
' archive and keeps the newest ones. The settings are the
' [one_click_backup] part of work.conf.
'
'   jdbasic one_click_backup.jdb                back up now
'   jdbasic one_click_backup.jdb --dry-run    show what would happen

IMPORT WORKCONF, BACKUP

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "one_click_backup."
DIM listed = WORKCONF.VALUE(cfg, key$ + "folders", ["~/Documents"])
DIM folders = SELECT(LAMBDA f$ -> WORKCONF.EXPAND$(f$), listed)
DIM home$ = "~/Documents/AutomateWork/backups"
DIM target$ = WORKCONF.FOLDER$(cfg, key$ + "target", home$)
```

```

DIM keep = WORKCONF.VALUE(cfg, key$ + "keep", 10)
DIM max_mb = WORKCONF.VALUE(cfg, key$ + "max_file_mb", 200)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM picked = BACKUP.ENTRIES(folders, max_mb)
DIM entries = picked{"entries"}
DIM skipped = picked{"skipped"}
DIM zip_name$ = BACKUP.ARCHIVE_NAME$("backup", NOW())
DIM zip_path$ = PATH.JOIN$(target$, zip_name$)
DIM k = 0
FOR k = 0 TO LEN(skipped) - 1
    PRINT "too large, left out: "; skipped[k]
NEXT k
IF dry_run THEN
    PRINT "would write "; zip_name$; " with "; LEN(entries); " files"
ELSE
    DIM n_files = BACKUP.WRITE(entries, zip_path$)
    PRINT "wrote "; zip_path$; " with "; n_files; " files"
ENDIF
DIM n_old = BACKUP.PRUNE(BACKUP.OLD(target$, "backup", keep), dry_run)
IF n_old > 0 THEN PRINT n_old; " older archives cleaned up."

```

The backup module collects the files, writes the archive and finds the old archives:

```

' backup.jdb: copies of folders as one dated ZIP archive, and the
' clean up that keeps only the newest archives.
'
'   IMPORT BACKUP
'   DIM picked = BACKUP.ENTRIES(["C:/Work"], 200)
'   BACKUP.WRITE(picked{"entries"}, "D:/Backup/backup 2026.zip")
'   BACKUP.PRUNE(BACKUP.OLD("D:/Backup", "backup", 10), FALSE)

EXPORT MODULE BACKUP

' TRUE for an office lock file such as "~$report.docx".
FUNC LockFile(file_name$)
    RETURN STARTSWITH(file_name$, "~$")
ENDFUNC

' The files under a folder, at any depth, as paths relative to it
' with forward slashes.
EXPORT FUNC FILES(root$)
    DIM out = []
    DIM entries = SORT(DIR$(PATH.JOIN$(root$, "*")))
    DIM k = 0
    DIM j = 0

```

```

    FOR k = 0 TO LEN(entries) - 1
        DIM file_name$ = entries[k]
        DIM full$ = PATH.JOIN$(root$, file_name$)
        IF FILE.ISDIR(full$) THEN
            DIM inner = FILES(full$)
            FOR j = 0 TO LEN(inner) - 1
                DIM sub_path$ = inner[j]
                PUSH(out, file_name$ + "/" + sub_path$)
            NEXT j
        ELSEIF NOT LockFile(file_name$) THEN
            PUSH(out, file_name$)
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' What goes into the archive: "entries" maps each name in the archive
' (the folder's own name, then the path inside it) to the file, and
' "skipped" lists files larger than max_mb.
EXPORT FUNC ENTRIES(folders, max_mb)
    DIM entries AS MAP
    DIM skipped = []
    DIM limit = max_mb * 1048576
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(folders) - 1
        DIM root$ = folders[k]
        IF NOT FILE.ISDIR(root$) THEN THROW "No such folder: " + root$
        DIM top$ = PATH.BASENAME$(root$)
        DIM rel = FILES(root$)
        FOR j = 0 TO LEN(rel) - 1
            DIM inside$ = rel[j]
            DIM full$ = PATH.JOIN$(root$, inside$)
            IF FILE.SIZE(full$) > limit THEN
                PUSH(skipped, full$)
            ELSE
                entries{top$ + "/" + inside$} = full$
            ENDIF
        NEXT j
    NEXT k
    RETURN {"entries": entries, "skipped": skipped}
ENDFUNC

' The archive's file name for a moment, such as
' "backup 2026-10-01 1830.zip".
EXPORT FUNC ARCHIVE_NAME$(prefix$, when)
    RETURN prefix$ + " " + FORMAT_DATE(when, "%Y-%m-%d %H%M") + ".zip"
ENDFUNC

```

```

' Writes the archive; answers the number of files in it.
EXPORT FUNC WRITE(entries, zip_path$)
    DIM content AS MAP
    DIM names = MAP.KEYS(entries)
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM entry$ = names[k]
        content{entry$} = BINREADER$(entries{entry$})
    NEXT k
    MKDIR PATH.DIRNAME$(zip_path$)
    RETURN ZIP.WRITE(zip_path$, content)
ENDFUNC

' The archives of a prefix in the target folder beyond the newest
' keep ones, oldest first.
EXPORT FUNC OLD(target$, prefix$, keep)
    DIM ours = []
    DIM entries = SORT(DIR$(PATH.JOIN$(target$, "*")))
    DIM k = 0
    FOR k = 0 TO LEN(entries) - 1
        DIM file_name$ = entries[k]
        DIM mine = STARTSWITH(file_name$, prefix$ + " ")
        IF mine ANDALSO ENDWITH(LCASE$(file_name$), ".zip") THEN
            PUSH(ours, PATH.JOIN$(target$, file_name$))
        ENDIF
    NEXT k
    IF LEN(ours) <= keep THEN RETURN []
    RETURN TAKE(LEN(ours) - keep, ours)
ENDFUNC

' Deletes the archives and prints a line for each; with dry_run it
' only prints them. Answers the number of archives in the list.
EXPORT FUNC PRUNE(old, dry_run)
    DIM k = 0
    FOR k = 0 TO LEN(old) - 1
        DIM path$ = old[k]
        IF dry_run THEN
            PRINT "would delete "; PATH.BASENAME$(path$)
        ELSE
            KILL path$
            PRINT "deleted "; PATH.BASENAME$(path$)
        ENDIF
    NEXT k
    RETURN LEN(old)
ENDFUNC

```

How it works

1. FILES walks a folder and all the folders inside it. It calls itself for every folder it finds, and leaves out the lock files Office writes while a document is open (~\$report.docx).
2. ENTRIES names each file in the archive by its folder and its path inside it, so Documents/Offerers/2026/Miller.pdf comes back where it was. Files larger than max_file_mb are listed and left out.
3. WRITE reads every file and hands the lot to ZIP.WRITE, which compresses what gets smaller and stores the rest.
4. OLD lists the archives that start with backup and end with .zip, sorted by name, which is sorted by date. Everything before the newest keep archives goes, through PRUNE and KILL.

Run it

```
jdbasic one_click_backup.jdb --dry-run
would write backup 2026-10-02 1630.zip with 1240 files
would delete backup 2026-07-24 1630.zip
1 older archives cleaned up.
```

Schedule it

The wizard plans the One-Click Backup for Friday at 16:30. If the laptop sleeps then, Windows runs it at the next start.

Make it yours

```
[one_click_backup]
folders = ["~/Documents", "~/Desktop"]
target = "E:/Backups"
keep = 10
max_file_mb = 200
```

- **More folders:** add them to folders.
- **Keep more history:** keep = 30 is about half a year of Fridays.
- **Large videos:** raise max_file_mb, or keep videos in a folder you do not back up this way.

When it goes wrong

- **"No such folder":** a folder in the list was renamed or the drive is not plugged in. The archive is not written, so nothing half done is left behind.
- **The program takes long or runs out of memory:** the archive is built in memory. Back up the large folders in a second recipe run of their own, or lower `max_file_mb`.
- **Two archives in one minute:** the second replaces the first, since they get the same name.

Balance dividend

About 15 minutes a week of copying, and the quiet knowledge that last Friday's work exists twice.

E05 Break Reminder

The chore

When Mia is deep in a spreadsheet, two hours pass without her standing up. In the evening her neck tells her. She has tried a timer on her phone; it rings in meetings and she switches it off for good.

What you get

During your working hours, a small message asks you to stand up, stretch and drink some water, every 50 minutes. Before work, after work and at the weekend it stays quiet. One click on OK and it is gone.

Before you start

The setup wizard has run and the Break Reminder is switched on with your working hours.

The program

The program is started every ten minutes. It decides whether a break is due and only then shows the message:

```
' Break Reminder: asks you to stand up and drink some water, every
' so many minutes within your working hours. The wizard starts it
' every ten minutes; it shows a message only when one is due. The
' settings are the [break_reminder] part of work.conf.
'
'   jdbasic break_reminder.jdb           remind me if it is time
'   jdbasic break_reminder.jdb --dry-run  only say whether it is time

IMPORT WORKCONF, BREAKS

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "break_reminder."
DIM start$ = WORKCONF.VALUE(cfg, key$ + "start", "08:30")
DIM end$ = WORKCONF.VALUE(cfg, key$ + "end", "17:30")
DIM every = WORKCONF.VALUE(cfg, key$ + "every_minutes", 50)
DIM weekdays = WORKCONF.VALUE(cfg, key$ + "workdays_only", TRUE)
DIM text$ = "Time for a break: stand up, stretch, drink some water."
DIM message$ = WORKCONF.VALUE(cfg, key$ + "message", text$)
DIM home$ = "~/Documents/AutomateWork/logs/break_reminder.txt"
```

```

DIM state$ = WORKCONF.FOLDER$(cfg, key$ + "state_file", home$)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM last$ = ""
IF FILE.EXISTS(state$) THEN last$ = TRIM$(TXTREADER$(state$))
DIM moment = NOW()
DIM due = BREAKS.DUE(moment, last$, start$, end$, every, weekdays)
IF NOT due THEN
    PRINT "No break due."
ELSEIF dry_run THEN
    PRINT "A break is due. Run without --dry-run to be reminded."
ELSE
    MKDIR PATH.DIRNAME$(state$)
    TXTWRITER state$, BREAKS.STAMP$(moment)
    DIM button = MSGBOX(message$, 64, "Break Reminder")
ENDIF

```

The decision is in a module of its own, so it can be tested without a single message box:

```

' breaks.jdb: whether a break reminder is due. Within working hours on
' a workday, once every so many minutes after the last reminder.
'
'   IMPORT BREAKS
'   IF BREAKS.DUE(NOW(), last$, "08:30", "17:30", 50, TRUE) THEN ...

EXPORT MODULE BREAKS

' The minute of the day a time written as "HH:MM" stands for.
EXPORT FUNC MINUTES(hhmm$)
    DIM parts = SPLIT(TRIM$(hhmm$), ":")
    IF LEN(parts) <> 2 THEN THROW "Not a time of day: " + hhmm$
    RETURN CINT(parts[0]) * 60 + CINT(parts[1])
ENDFUNC

' TRUE when a moment lies inside working hours; with workdays_only
' Saturday and Sunday are outside.
EXPORT FUNC WORKING(when, start$, end$, workdays_only)
    DIM wday = CINT(FORMAT_DATE(when, "%w"))
    DIM weekend = (wday = 0) ORELSE (wday = 6)
    IF workdays_only ANDALSO weekend THEN RETURN FALSE
    DIM now_min = HOUR(when) * 60 + MINUTE(when)
    RETURN now_min >= MINUTES(start$) ANDALSO now_min < MINUTES(end$)
ENDFUNC

' TRUE when a reminder is due: inside working hours, and at least
' every_min minutes after the last one, or on a new day. last$ is ""

```

```
' before the first reminder, else "YYYY-MM-DD HH:MM:SS".
EXPORT FUNC DUE(when, last$, start$, end$, every_min, workdays_only)
    DIM inside = WORKING(when, start$, end$, workdays_only)
    IF NOT inside THEN RETURN FALSE
    IF last$ = "" THEN RETURN TRUE
    DIM last = CDATE(last$)
    DIM last_day$ = FORMAT_DATE(last, "%Y-%m-%d")
    IF last_day$ <> FORMAT_DATE(when, "%Y-%m-%d") THEN RETURN TRUE
    RETURN DATEDIFF("N", last, when) >= every_min
ENDFUNC

' The text of a moment as DUE expects it for the last reminder.
EXPORT FUNC STAMP$(when)
    RETURN FORMAT_DATE(when, "%Y-%m-%d %H:%M:%S")
ENDFUNC
```

How it works

1. The program remembers the time of the last reminder in a small text file in your work folder.
2. WORKING checks the time of day against your working hours, and the weekday: FORMAT_DATE(when, "%w") is 0 for Sunday and 6 for Saturday.
3. DUE answers yes when no reminder came yet today, or when the last one is at least every_min minutes old.
4. When a break is due, the program writes the new time first and then shows the message with MSGBOX. The 64 makes it the friendly kind with an information sign.

The scheduler starting the program every ten minutes is simpler than a program that runs all day: nothing to keep alive, nothing to restart after the laptop slept.

Run it

```
jdbasic break_reminder.jdb --dry-run
A break is due. Run without --dry-run to be reminded.
```

Schedule it

The wizard plans the Break Reminder every ten minutes. It decides on its own whether it is time, so the schedule does not need to know your working hours.

Make it yours

```
[break_reminder]
start = "08:30"
end = "17:30"
every_minutes = 50
workdays_only = true
message = "Time for a break: stand up, stretch, drink some water."
```

- **Longer stretches:** `every_minutes = 90`.
- **Your own words:** change message; the reminder that makes you smile is the one you follow.
- **Shift work:** set `workdays_only = false` and your hours.

When it goes wrong

- **No reminder at all:** run it with `--dry-run` during your working hours. "No break due" before 50 minutes have passed is right.
- **A reminder during a presentation:** Windows' focus assist hides most messages; switch it on while you present.
- **Reminders right after another:** two schedules start the program. Remove one in the wizard.

Balance dividend

No minutes saved, a few given back to your body: about eight short breaks a day.

E06 Shutdown Ritual

The chore

Mia's working day has no end. She closes the laptop at six, opens it again at nine to check one thing, and lies awake with the list of what is still open. On Monday she cannot remember what she finished on Thursday, and the weekly call with her team lead starts with ten minutes of scrolling through sent mail.

What you get

At the end of every working day the Shutdown Ritual asks three short questions: what got done, what comes first tomorrow, and how the day was from 1 to 5. It writes the answers into a journal file, one entry per day, and on Fridays it shows the week: how many days are written down, the average mood, everything that got done and what is still open. Then it tells you to close the laptop.

Before you start

The setup wizard has run and the Shutdown Ritual is switched on. The wizard planned it for the end of your working hours.

The program

The program asks the questions and writes the entry:

```
' Shutdown Ritual: three questions at the end of the day, the answers
' kept in a journal, and on the summary day a look back at the week.
' The settings are the [shutdown_ritual] part of work.conf.
'
'   jdbasic shutdown_ritual.jdb          ask, write, close the day
'   jdbasic shutdown_ritual.jdb --dry-run ask, show, write nothing

IMPORT WORKCONF, RITUAL

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "shutdown_ritual."
DIM fallback$ = "~/Documents/AutomateWork/journal/journal.md"
DIM journal$ = WORKCONF.FOLDER$(cfg, key$ + "journal", fallback$)
DIM summary_day = WORKCONF.VALUE(cfg, key$ + "summary_weekday", 5)
DIM dry_run = WORKCONF.FLAG("dry-run")
DIM today$ = FORMAT_DATE(NOW(), "%Y-%m-%d")
```

```

DIM done$ = ""
DIM tomorrow$ = ""
DIM mood$ = ""
PRINT "Time to close the day."
INPUT "What did you get done today? "; done$
INPUT "What is the first thing for tomorrow? "; tomorrow$
DIM mood = 0
DO WHILE mood = 0
    INPUT "How was the day, from 1 (hard) to 5 (good)? "; mood$
    mood = CINT(RITUAL.MOOD(mood$))
LOOP

DIM entry$ = RITUAL.ENTRY$(today$, done$, tomorrow$, mood)
IF dry_run THEN
    PRINT "This would go into "; journal$; ":"
    PRINT entry$
ELSE
    RITUAL.ADD(journal$, entry$)
    PRINT "Written to "; journal$
ENDIF

IF RITUAL.WEEKDAY_OF(today$) = summary_day THEN
    DIM so_far$ = RITUAL.TEXT_OF$(journal$)
    IF dry_run THEN so_far$ = so_far$ + entry$
    DIM entries = RITUAL.PARSE(so_far$)
    PRINT
    PRINT RITUAL.SUMMARY$(RITUAL.WEEK(entries, today$))
ENDIF
PRINT "Done for today. Close the laptop."

```

The module knows how an entry looks, how to read the journal back and how to sum up a week:

```

' ritual.jdb: the end of the working day as a journal. An entry holds
' what got done, what is open for tomorrow and a mood from 1 to 5;
' the entries of a week read back as a short summary.
'
'   IMPORT RITUAL
'   DIM e$ = RITUAL.ENTRY$("2026-10-02", "offer sent", "call", 4)
'   RITUAL.ADD("journal.md", e$)
'   PRINT RITUAL.SUMMARY$(RITUAL.WEEK(RITUAL.ENTRIES("journal.md"), _
'       "2026-10-02"))

EXPORT MODULE RITUAL

DIM gLF$ = CHR$(10)

```

```

' The weekday number of a date given as YYYY-MM-DD, 0 for Sunday.
EXPORT FUNC WEEKDAY_OF(day$)
    RETURN VAL(FORMAT_DATE(CDATE(day$), "%w"))
ENDFUNC

' The Monday of the week a day belongs to, as YYYY-MM-DD.
EXPORT FUNC MONDAY$(day$)
    DIM back = (WEEKDAY_OF(day$) + 6) MOD 7
    RETURN FORMAT_DATE(DATEADD("D", -back, CDATE(day$)), "%Y-%m-%d")
ENDFUNC

' A mood answer as a number from 1 to 5, or 0 when it is none.
EXPORT FUNC MOOD(answer$)
    DIM v = VAL(TRIM$(answer$))
    IF v < 1 ORELSE v > 5 ORELSE v <> INT(v) THEN RETURN 0
    RETURN v
ENDFUNC

' One journal entry as text.
EXPORT FUNC ENTRY$(day$, done$, tomorrow$, mood)
    DIM weekday$ = FORMAT_DATE(CDATE(day$), "%A")
    DIM out$ = "## " + day$ + " " + weekday$ + gLF$
    out$ = out$ + "Mood: " + STR$(mood) + " of 5" + gLF$
    out$ = out$ + "Done: " + TRIM$(done$) + gLF$
    out$ = out$ + "Tomorrow: " + TRIM$(tomorrow$) + gLF$
    RETURN out$
ENDFUNC

' Adds an entry to the end of the journal file, creating its folder.
EXPORT SUB ADD(journal$, entry$)
    DIM folder$ = PATH.DIRNAME$(journal$)
    IF folder$ <> "" THEN MKDIR folder$
    TXTWRITER journal$, entry$ + gLF$, TRUE
ENDSUB

' The text of a journal file, "" when there is none yet.
EXPORT FUNC TEXT_OF$(journal$)
    IF NOT FILE.EXISTS(journal$) THEN RETURN ""
    RETURN TXTREADER$(journal$)
ENDFUNC

' The entries of a journal file, oldest first.
EXPORT FUNC ENTRIES(journal$)
    RETURN PARSE(TEXT_OF$(journal$))
ENDFUNC

' The entries of journal text, oldest first: maps with "day", "mood",

```



```

' "done" and "tomorrow".
EXPORT FUNC PARSE(journal_text$)
  DIM out = []
  DIM rows = SPLIT(REPLACE$(journal_text$, CHR$(13), ""), gLF$)
  DIM cur = {"day": ""}
  DIM k = 0
  FOR k = 0 TO LEN(rows) - 1
    DIM row$ = rows[k]
    IF STARTSWITH(row$, "## ") THEN
      IF cur{"day"} <> "" THEN PUSH(out, cur)
      DIM day$ = MID$(row$, 3, 10)
      cur = {"day": day$, "mood": 0, "done": "", "tomorrow": ""}
    ELSEIF STARTSWITH(row$, "Mood: ") THEN
      cur{"mood"} = VAL(MID$(row$, 6, 2))
    ELSEIF STARTSWITH(row$, "Done: ") THEN
      cur{"done"} = MID$(row$, 6, LEN(row$))
    ELSEIF STARTSWITH(row$, "Tomorrow: ") THEN
      cur{"tomorrow"} = MID$(row$, 10, LEN(row$))
    ENDIF
  NEXT k
  IF cur{"day"} <> "" THEN PUSH(out, cur)
  RETURN out
ENDFUNC

' The entries from the Monday to the Sunday of the week of day$.
EXPORT FUNC WEEK(entries, day$)
  DIM first$ = MONDAY$(day$)
  DIM sunday = DATEADD("D", 6, CDATE(first$))
  DIM last$ = FORMAT_DATE(sunday, "%Y-%m-%d")
  RETURN FILTER(LAMBDA USE(first$, last$) e -> _
    e{"day"} >= first$ AND ALSO e{"day"} <= last$, entries)
ENDFUNC

' A summary of a week's entries: days written, the average mood, the
' things done and what is still open from the last day.
EXPORT FUNC SUMMARY$(week)
  IF LEN(week) = 0 THEN RETURN "No entries this week."
  DIM all_moods = SELECT(LAMBDA e -> e{"mood"}, week)
  DIM moods = FILTER(LAMBDA m -> m > 0, all_moods)
  DIM written = FILTER(LAMBDA e -> e{"done"} <> "", week)
  DIM done = SELECT(LAMBDA e -> "-" + e{"done"}, written)
  DIM out$ = "Week of " + MONDAY$(week[0]{"day"}) + ": "
  out$ = out$ + STR$(LEN(week)) + " days in the journal"
  IF LEN(moods) > 0 THEN
    DIM avg$ = FORMAT$("{:.1f}", SUM(moods) / LEN(moods))
    out$ = out$ + ", mood " + avg$ + " on average"
  ENDIF
  out$ = out$ + "." + gLF$ + "Done:" + gLF$ + JOIN(done, gLF$)

```

```

DIM open$ = week[LEN(week) - 1]{"tomorrow"}
IF open$ <> "" THEN out$ = out$ + gLF$ + "Still open: " + open$
RETURN out$
ENDFUNC

```

How it works

1. INPUT shows a question and waits for the answer. The mood question repeats until the answer is a whole number from 1 to 5; MOOD answers 0 for anything else.
2. ENTRY\$ turns the answers into four lines: a heading with the date and the weekday, the mood, what got done and what comes next.
3. ADD appends the entry to the journal with TXTWRITER and its third argument TRUE, which adds to the end of the file instead of replacing it. The journal is a plain text file you can open in any editor.
4. On the summary day PARSE reads the journal back into a list of entries, and WEEK keeps the ones from Monday to Sunday of this week with FILTER. Its lambda compares each day with the first and the last day of the week; USE(first\$, last\$) hands it those two values of the function.
5. SUMMARY\$ writes the look back. SELECT takes the mood out of every entry and FILTER drops the days without one, so the average is SUM divided by LEN. The same pair collects what got done.
6. MONDAY\$ finds the start of the week from the weekday number that FORMAT_DATE gives with %w: 0 for Sunday, 1 for Monday, and so on.

Run it

```

jdbasic shutdown_ritual.jdb
Time to close the day.
What did you get done today? sent the Miller offer
What is the first thing for tomorrow? call the printer
How was the day, from 1 (hard) to 5 (good)? 4
Written to C:\Users\mia\Documents\AutomateWork\journal\journal.md
Done for today. Close the laptop.

```

With `--dry-run` it asks the same questions and shows the entry, but writes nothing.

Schedule it

The wizard plans the ritual on weekdays at the end of the working hours you gave it. A console window opens with the first question. You can also run it by hand at any time.

Make it yours

```
[shutdown_ritual]
journal = "~/Documents/AutomateWork/journal/journal.md"
summary_weekday = 5
```

- **Another summary day:** `summary_weekday = 1` shows the week on Monday morning instead, which some people like better for planning.
- **A journal per year:** point `journal` at a file such as `journal 2026.md` and change it in January.
- **Other questions:** the three `INPUT` lines in the program are the questions. Change their wording, and keep the order, since the entry stores the answers by position.

When it goes wrong

- **"is a Windows path in double quotes":** the program stops before the first question and names the line in `work.conf`. Inside double quotes a backslash starts a special character in a TOML file, so `"C:\Users\mia"` cannot stay as written. Write the path with forward slashes, as above, or in single quotes, which keep backslashes: `'C:\Users\mia\journal.md'`.
- **No summary on Friday:** the summary counts the days of this week only. Entries from last week stay in the journal but do not appear.
- **The window closes too fast:** start the ritual from a console window; it stays open after the last line.

Balance dividend

About 20 minutes a week: no searching for what you did when someone asks, and a clear point where the working day ends.

E07 Time Tracker

The chore

Mia works for four projects at once, and every Friday the office asks how her hours split between them. She rebuilds the week from her calendar and her memory, guesses the rest, and spends half an hour on a number nobody fully trusts.

What you get

Two short commands mark when work on a project starts and stops. The Time Tracker writes each of them as a line in a log file, and on Friday it turns the week into an Excel sheet: one row per project, one column per day, the totals at the end.

Before you start

The setup wizard has run and the Time Tracker is switched on. The wizard planned the weekly sheet for Friday afternoon; starting and stopping is up to you.

The program

The program reads the command and does one of four things:

```
' Time Tracker: start and stop the clock for a project, see what runs,
' and turn the week into an Excel sheet. The settings are the
' [time_tracker] part of work.conf.
'
'   jdbasic time_tracker.jdb start "Miller offer"
'   jdbasic time_tracker.jdb stop
'   jdbasic time_tracker.jdb status
'   jdbasic time_tracker.jdb week [--dry-run]

IMPORT WORKCONF, TRACKER

' The words on the command line that are no --switch and no value
' of --config.
FUNC Words()
    DIM argv = OS.ARGV()
    DIM out = []
```

```

    DIM k = 1
    DO WHILE k < LEN(argv)
        DIM a$ = argv[k]
        IF a$ = "--config" THEN
            k = k + 2
        ELSE
            IF NOT STARTSWITH(a$, "--") THEN PUSH(out, a$)
            k = k + 1
        ENDIF
    LOOP
    RETURN out
ENDFUNC

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "time_tracker."
DIM home$ = "~/Documents/AutomateWork/time"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM log_path$ = PATH.JOIN$(folder$, "log.csv")
DIM now$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M:%S")
DIM words = Words()
DIM command$ = "status"
IF LEN(words) > 0 THEN command$ = LCASE$(words[0])
DIM work = TRACKER.SESSIONS(TRACKER.EVENTS(log_path$))
DIM running$ = TRACKER.RUNNING$(work)

IF command$ = "start" THEN
    IF LEN(words) < 2 THEN THROW "start needs a project name"
    DIM project$ = TRACKER.CLEAN$(JOIN(DROP(1, words), " "))
    TRACKER.RECORD(log_path$, now$, "start", project$)
    IF running$ <> "" THEN PRINT "Stopped "; running$; "."
    PRINT "Started "; project$; " at "; MID$(now$, 11, 5); "."
ELSEIF command$ = "stop" THEN
    IF running$ = "" THEN
        PRINT "Nothing is running."
    ELSE
        TRACKER.RECORD(log_path$, now$, "stop", "")
        PRINT "Stopped "; running$; "."
    ENDIF
ELSEIF command$ = "status" THEN
    IF running$ = "" THEN
        PRINT "Nothing is running."
    ELSE
        DIM since$ = work[LEN(work) - 1]{"start"}
        DIM mins = TRACKER.MINUTES(since$, now$)
        PRINT running$; " runs for "; FLOOR(mins); " minutes."
    ENDIF
ELSEIF command$ = "week" THEN
    DIM monday$ = TRACKER.MONDAY$(LEFT$(now$, 10))

```

```

DIM table = TRACKER.WEEK_TABLE(work, monday$, now$)
DIM rows = TRACKER.SHEET_ROWS(table, monday$)
DIM k = 0
FOR k = 1 TO LEN(rows) - 1
    DIM row = rows[k]
    DIM hours$ = FORMAT$("{:.2f}", row[LEN(row) - 1])
    PRINT row[0]; ": "; hours$; " h"
NEXT k
IF NOT WORKCONF.FLAG("dry-run") THEN
    DIM sheet$ = PATH.JOIN$(folder$, "week " + monday$ + ".xlsx")
    TRACKER.WRITE_SHEET(rows, sheet$)
    PRINT "Wrote "; sheet$
ENDIF
ELSE
    PRINT "Use start <project>, stop, status or week."
ENDIF

```

The module keeps the log, turns it into blocks of work and adds up the hours:

```

' tracker.jdb: time per project from a log of start and stop events.
' The log is a CSV file with the columns time, event and project; a
' line is added for every start and stop, nothing is ever rewritten.
'
'   IMPORT TRACKER
'   TRACKER.RECORD("log.csv", now$, "start", "Miller")
'   DIM work = TRACKER.SSESSIONS(TRACKER.EVENTS("log.csv"))
'   DIM week = TRACKER.WEEK_TABLE(work, "2026-09-28", now$)

EXPORT MODULE TRACKER

IMPORT XLSX

DIM gLF$ = CHR$(10)

' A project name on one line: line breaks become spaces.
EXPORT FUNC CLEAN$(project$)
    DIM p$ = REPLACE$(project$, CHR$(13), " ")
    RETURN TRIM$(REPLACE$(p$, gLF$, " "))
ENDFUNC

' A name as one CSV field: in quotes when it holds a comma or a quote,
' with a quote inside written twice.
FUNC Field$(name$)
    DIM q$ = CHR$(34)
    DIM plain = INSTR(name$, ",") < 0 AND ALSO INSTR(name$, q$) < 0
    IF plain THEN RETURN name$
    RETURN q$ + REPLACE$(name$, q$, q$ + q$) + q$

```

```

ENDFUNC

' Adds one event ("start" or "stop") to the log, with a header line
' when the log is new.
EXPORT SUB RECORD(log_path$, when$, event$, project$)
    IF NOT FILE.EXISTS(log_path$) THEN
        DIM folder$ = PATH.DIRNAME$(log_path$)
        IF folder$ <> "" THEN MKDIR folder$
        TXTWRITER log_path$, "time,event,project" + gLF$
    ENDIF
    DIM row$ = when$ + "," + event$ + "," + Field$(CLEAN$(project$))
    TXTWRITER log_path$, row$ + gLF$, TRUE
ENDSUB

' The events of the log as rows of [time, event, project].
EXPORT FUNC EVENTS(log_path$)
    IF NOT FILE.EXISTS(log_path$) THEN RETURN []
    DIM kinds = ["STRING", "STRING", "STRING"]
    RETURN CSVREADER(log_path$, ",", TRUE, kinds)
ENDFUNC

' The blocks of work the events describe: maps with "project",
' "start" and "stop" ("" while it is still running). A start while
' another block runs ends that one first.
EXPORT FUNC SESSIONS(events)
    DIM out = []
    DIM in_block = FALSE
    DIM cur = {"project": "", "start": "", "stop": ""}
    DIM k = 0
    FOR k = 0 TO LEN(events) - 1
        DIM ev = events[k]
        DIM when$ = ev[0]
        DIM what$ = ev[1]
        IF in_block THEN
            cur{"stop"} = when$
            PUSH(out, cur)
            in_block = FALSE
        ENDIF
        IF what$ = "start" THEN
            DIM project$ = ""
            IF LEN(ev) > 2 THEN project$ = ev[2]
            cur = {"project": project$, "start": when$, "stop": ""}
            in_block = TRUE
        ENDIF
    NEXT k
    IF in_block THEN PUSH(out, cur)
    RETURN out
ENDFUNC

```

```

' The project of the block that is still running, "" when none is.
EXPORT FUNC RUNNING$(sessions)
    IF LEN(sessions) = 0 THEN RETURN ""
    DIM last = sessions[LEN(sessions) - 1]
    IF last{"stop"} = "" THEN RETURN last{"project"}
    RETURN ""
ENDFUNC

' The Monday of the week a day (YYYY-MM-DD) belongs to.
EXPORT FUNC MONDAY$(day$)
    DIM back = (VAL(FORMAT_DATE(CDATE(day$), "%w")) + 6) MOD 7
    RETURN FORMAT_DATE(DATEADD("D", -back, CDATE(day$)), "%Y-%m-%d")
ENDFUNC

' The minutes between two times written YYYY-MM-DD HH:MM:SS.
EXPORT FUNC MINUTES(start$, stop$)
    RETURN DATEDIFF("N", CDATE(start$), CDATE(stop$))
ENDFUNC

' Hours per project and weekday for the week from monday$: a map from
' project to seven numbers, Monday first. A block counts on the day
' it started; a running block counts until now$.
EXPORT FUNC WEEK_TABLE(sessions, monday$, now$)
    DIM table AS MAP
    DIM k = 0
    FOR k = 0 TO LEN(sessions) - 1
        DIM s = sessions[k]
        DIM day$ = LEFT$(s{"start"}, 10)
        DIM col = DATEDIFF("D", CDATE(monday$), CDATE(day$))
        IF col >= 0 AND ALSO col <= 6 THEN
            DIM stop$ = s{"stop"}
            IF stop$ = "" THEN stop$ = now$
            DIM project$ = s{"project"}
            IF NOT MAP.EXISTS(table, project$) THEN
                table{project$} = ZEROS(7)
            ENDIF
            DIM hours = table{project$}
            hours[col] = hours[col] + MINUTES(s{"start"}, stop$) / 60
            table{project$} = hours
        ENDIF
    NEXT k
    RETURN table
ENDFUNC

' Hours rounded to two decimals.
FUNC Hours2(h)
    RETURN ROUND(h * 100) / 100

```



```

ENDFUNC

' The week as rows for a sheet: a header with the days, one row per
' project with its total, and a row with the totals of every day.
EXPORT FUNC SHEET_ROWS(table, monday$)
    DIM head = ["Project"]
    DIM k = 0
    DIM d = 0
    FOR k = 0 TO 6
        DIM day = DATEADD("D", k, CDATE(monday$))
        PUSH(head, FORMAT_DATE(day, "%a %d.%m."))
    NEXT k
    PUSH(head, "Total")
    DIM rows = [head]
    DIM sums = ZEROS(7)
    DIM projects = SORT(MAP.KEYS(table))
    FOR k = 0 TO LEN(projects) - 1
        DIM p$ = projects[k]
        DIM hours = table[p$]
        DIM row = [p$]
        DIM total = 0
        FOR d = 0 TO 6
            PUSH(row, Hours2(hours[d]))
            total = total + hours[d]
            sums[d] = sums[d] + hours[d]
        NEXT d
        PUSH(row, Hours2(total))
        PUSH(rows, row)
    NEXT k
    DIM last = ["Total"]
    FOR d = 0 TO 6
        PUSH(last, Hours2(sums[d]))
    NEXT d
    PUSH(last, Hours2(SUM(sums)))
    PUSH(rows, last)
    RETURN rows
ENDFUNC

' Writes the rows as an Excel sheet with the hours in two decimals.
EXPORT SUB WRITE_SHEET(rows, path$)
    DIM folder$ = PATH.DIRNAME$(path$)
    IF folder$ <> "" THEN MKDIR folder$
    DIM wb = XLSX.NEW()
    DIM sheet = XLSX.SHEET(wb, "Week", rows)
    XLSX.COLUMN(sheet, "A", {"width": 24})
    DIM c = 2
    FOR c = 2 TO 9
        XLSX.COLUMN(sheet, c, {"width": 11, "format": "0.00"})
    NEXT c

```

```

NEXT c
DIM last_row$ = STR$(LEN(rows))
XLSX.STYLE(sheet, "A" + last_row$ + ":I" + last_row$, _
    {"bold": TRUE, "fill": "DDEBF7"})
XLSX.FREEZE(sheet, 1)
XLSX.WRITE(path$, wb)
ENDSUB

```

How it works

1. RECORD adds one line to the log, such as 2026-10-01 09:00:00,start,Miller offer. The log only ever grows, so nothing is lost when the program stops halfway. A name with a comma goes in quotes, "Miller, offer", so it stays one field.
2. EVENTS reads the log with CSVREADER. The list of column types keeps every field as text, so a date stays a date as written.
3. SESSIONS pairs the events into blocks of work. A start while another project runs ends that one first, so switching projects is a single command.
4. WEEK_TABLE adds up the minutes per project and day from Monday on, with DATEDIFF in minutes. A block that still runs counts until now.
5. SHEET_ROWS lays the table out with a header row, the totals per project in the last column and the totals per day in the last row; WRITE_SHEET saves it with the XLSX library, hours in two decimals.

Run it

```

jdbasic time_tracker.jdb start Miller offer
Started Miller offer at 09:02.
jdbasic time_tracker.jdb start Admin
Stopped Miller offer.
Started Admin at 11:30.
jdbasic time_tracker.jdb week
Admin: 0.50 h
Miller offer: 2.47 h
Total: 2.97 h
Wrote C:\Users\mia\Documents\AutomateWork\time\week 2026-09-28.xlsx

```

status tells you what runs and for how long. week --dry-run shows the totals without writing the sheet.

Schedule it

The wizard plans `time_tracker.jdb` week for Friday at 16:00, so the sheet is ready before the week ends. The start and stop commands are yours: put them on desktop shortcuts, one per project you work on often.

Make it yours

```
[time_tracker]
folder = "~/Documents/AutomateWork/time"
```

- **A shared folder:** point `folder` at a folder your team lead can open, and the weekly sheets land there on their own.
- **Project names with spaces** work as they are: start Miller offer starts "Miller offer".
- **Open the log in Excel:** `log.csv` is a plain CSV file. Correct a forgotten stop there, then run week again.

When it goes wrong

- **A block that lasted all night:** you forgot stop. Edit the time of the next line in `log.csv`, or add a stop line with the right time.
- **A block over midnight** counts on the day it started.

Balance dividend

About 30 minutes a week of rebuilding your hours, and numbers you can stand behind when someone asks.

E08 Mail Templates

The chore

Mia writes the same five mails every week: thanks for an order, the notes after a meeting, the note that she is away. Each time she looks for last week's version in her sent mail, copies it, changes the name and the date, and once a month sends one with the wrong name in it.

What you get

Your standard replies live as small text files with placeholders such as {{ to }} and {{ date }}. One command fills a template with the name of the person it goes to, today's date and your own values, shows the mail and puts it on the clipboard. You paste it into your mail program and send it.

Before you start

The setup wizard has run, copied three example templates into Documents\AutomateWork\templates and asked for your name. There is nothing to schedule: you run this recipe when you write a mail.

The program

The program fills the template you name and copies the result:

```
' Mail Templates: a standard reply filled with a name, the date and
' your own values, put on the clipboard to paste into a mail. The
' settings are the [mail_templates] part of work.conf.
'
'   jdbasic mail_templates.jdb                list the templates
'   jdbasic mail_templates.jdb thanks Ms Miller  fill and copy one
'   jdbasic mail_templates.jdb thanks --dry-run  only show it

IMPORT WORKCONF, MAILTPL

' The words on the command line that are no --switch and no value
' of --config.
FUNC Words()
    DIM argv = OS.ARGS()
    DIM out = []
    DIM k = 1
```

```

DO WHILE k < LEN(argv)
    DIM a$ = argv[k]
    IF a$ = "--config" THEN
        k = k + 2
    ELSE
        IF NOT STARTSWITH(a$, "--") THEN PUSH(out, a$)
        k = k + 1
    ENDIF
LOOP
RETURN out
ENDFUNC

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "mail_templates."
DIM home$ = "~/Documents/AutomateWork/templates"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM me$ = WORKCONF.VALUE(cfg, key$ + "me", "")
DIM sign$ = WORKCONF.VALUE(cfg, key$ + "signature", "Kind regards")
DIM own AS MAP
DIM extra = WORKCONF.VALUE(cfg, key$ + "values", own)
DIM words = Words()

IF LEN(words) = 0 THEN
    PRINT "Templates in "; folder$; ":"
    PRINT " "; JOIN(MAILTPL.NAMES(folder$), ", ")
ELSE
    DIM to$ = JOIN(DROP(1, words), " ")
    DIM values = MAILTPL.VALUES(me$, sign$, to$, NOW())
    DIM names = MAP.KEYS(extra)
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        values{names[k]} = extra{names[k]}
    NEXT k
    DIM template$ = MAILTPL.TEMPLATE$(folder$, words[0])
    DIM gaps = MAILTPL.MISSING(template$, values)
    IF LEN(gaps) > 0 THEN PRINT "No value for: "; JOIN(gaps, ", ")
    DIM mail$ = MAILTPL.FILL$(template$, values)
    PRINT mail$
    IF NOT WORKCONF.FLAG("dry-run") THEN
        CLIPBOARD.SET mail$
        PRINT "(on the clipboard, paste it into your mail)"
    ENDIF
ENDIF
ENDIF

```

The module finds the templates, fills them and turns the result into plain text:

```

' mailtpl.jdb: standard replies from template files. A template is a
' text file with {{ placeholders }}; it is filled with the name of the
' person it goes to, the date and the values of work.conf, as plain
' text ready to paste into a mail.
'
'   IMPORT MAILTPL
'   DIM v = MAILTPL.VALUES("Mia", "Kind regards", "Ms Miller", NOW())
'   PRINT MAILTPL.FILL$(MAILTPL.TEMPLATE$("templates", "thanks"), v)

EXPORT MODULE MAILTPL

IMPORT TMPL

' The template names in a folder: its .txt files without the ending.
EXPORT FUNC NAMES(folder$)
    DIM out = []
    IF NOT FILE.ISDIR(folder$) THEN RETURN out
    DIM files = SORT(DIR$(PATH.JOIN$(folder$, "*.txt")))
    RETURN SELECT(LAMBDA f$ -> LEFT$(f$, LEN(f$) - 4), files)
ENDFUNC

' The text of the template called name$ in the folder.
EXPORT FUNC TEMPLATE$(folder$, name$)
    DIM path$ = PATH.JOIN$(folder$, name$ + ".txt")
    IF NOT FILE.EXISTS(path$) THEN
        DIM have$ = JOIN(NAMES(folder$), ", ")
        THROW "No template " + name$ + ". There are: " + have$
    ENDIF
    RETURN REPLACE$(TXTREADER$(path$), CHR$(13), "")
ENDFUNC

' The values a template can use: me, signature, to, date and weekday.
EXPORT FUNC VALUES(me$, signature$, to$, today)
    DIM v AS MAP
    v{"me"} = me$
    v{"signature"} = signature$
    v{"to"} = to$
    DIM day$ = FORMAT_DATE(today, "%d")
    IF LEFT$(day$, 1) = "0" THEN day$ = MID$(day$, 1, 1)
    v{"date"} = day$ + FORMAT_DATE(today, " %B %Y")
    v{"weekday"} = FORMAT_DATE(today, "%A")
    RETURN v
ENDFUNC

' Text with the HTML entities of the template engine turned back into
' the characters they stand for.
EXPORT FUNC PLAIN$(t$)
    DIM out$ = REPLACE$(REPLACE$(t$, "<", "<"), ">", ">")

```

```

        out$ = REPLACE$(REPLACE$(out$, "&quot;", CHR$(34)), "&#39;", "'")
        RETURN REPLACE$(out$, "&amp;", "&")
    ENDFUNC

' The template filled with the values, as plain text.
EXPORT FUNC FILL$(template$, values)
    RETURN PLAIN$(TMPL.RENDERSTR$(template$, values))
ENDFUNC

' The placeholders of a template that have no value, each once.
EXPORT FUNC MISSING(template$, values)
    DIM found = REGEX.FINDALL("\{\{\s*([A-Za-z_]+)", template$)
    DIM seen AS MAP
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(found) - 1
        DIM key$ = found[k][0]
        DIM given = MAP.EXISTS(values, key$)
        IF NOT given AND ALSO NOT MAP.EXISTS(seen, key$) THEN
            PUSH(out, key$)
            seen{key$} = TRUE
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

```

One of the templates that come with the book:

```

Dear {{ to }},

thank you for your order. We have it in our system and will confirm
the delivery date by {{ weekday }} at the latest.

{{ signature }}
{{ me }}

```

How it works

1. NAMES lists the .txt files of the templates folder and takes the ending off every name in one line, with SELECT and a LAMBDA as in Chapter 2. Without a template name the program shows that list.
2. VALUES collects what a template can use: me and signature from work.conf, to from the command line, and the date written out, such as 2 October 2026, with its weekday.

3. Your own values from `work.conf` join them, so a template can use any name you define there.
4. `MISSING` warns about placeholders that have no value, before the mail goes out with a gap in it.
5. `FILL$` lets the `TMPL` library fill the template. `TMPL` was written for web pages and writes & as &; `PLAIN$` turns those back, since a mail is plain text.
6. `CLIPBOARD.SET` puts the mail on the clipboard.

Run it

```
jdbasic mail_templates.jdb thanks Ms Miller
Dear Ms Miller,

thank you for your order. We have it in our system and will confirm
the delivery date by Friday at the latest.

Kind regards
Mia Schmidt
(on the clipboard, paste it into your mail)
```

Without a template name it lists the templates; with `--dry-run` it shows the mail and leaves the clipboard alone.

Schedule it

Nothing to schedule. Put the command on a desktop shortcut for the template you use most, or run it from a console window.

Make it yours

```
[mail_templates]
folder = "~/Documents/AutomateWork/templates"
me = "Mia Schmidt"
signature = "Kind regards"

[mail_templates.values]
back = "Monday, 12 October"
deputy = "Jonas Becker"
```


- **A new template:** save a text file such as `invoice.txt` in the templates folder. Its name is the command: `mail_templates.jdb invoice Ms Miller`.
- **Your own placeholders:** every line under `[mail_templates.values]` becomes a placeholder. The away note uses `{{ back }}` and `{{ deputy }}`.
- **Different greetings:** a template can choose with a condition, such as `{% if to %}Dear {{ to }},{% else %}Hello,{% endif %}`.

When it goes wrong

- **"No value for: back, deputy":** the template uses placeholders that `work.conf` does not define. Add them under `[mail_templates.values]`.
- **"No template invoice":** the message lists the templates there are; check the spelling and that the file ends in `.txt`.
- **The clipboard is empty:** some remote desktop sessions block it. The mail is printed as well; copy it from the window.

Balance dividend

About 25 minutes a week of searching and editing old mails, and no more replies that greet the wrong person.

E09 Meeting Notes Starter

The chore

Mia takes the notes in most project meetings. Before each one she opens Word, types the title of the meeting, looks up the time and the room in the calendar, copies the names of the attendees from the invitation and adds the headings she always uses. It takes five minutes, three or four times a week, and on a busy morning the meeting has started before the file is ready.

What you get

Every weekday morning the Meeting Notes Starter looks at your calendar and finds the next meeting. It writes a Word file named after the date and the title, such as 2026-10-05 Miller project status.docx, with the time, the place, the attendees, the agenda from the invitation and an empty table for the actions. When you walk into the room, the file is already in your notes folder.

Before you start

The recipe reads a calendar file in the iCalendar format (.ics). Outlook exports one with *File, Save Calendar*; Google Calendar gives you one under *Settings, Import and export*. Save it as calendar.ics in your work folder, or point the setting calendar at the file your calendar program keeps up to date.

The program

The program the wizard runs reads the settings, asks for the next meeting and writes the file, unless a file of that name is already there:

```
' Meeting Notes Starter: finds the next meeting in your calendar file
' and writes a Word file to take its notes in. The settings are the
' [meeting_notes] part of work.conf.
'
' jdbasic meeting_notes.jdb          write the notes file
' jdbasic meeting_notes.jdb --dry-run only show which meeting

IMPORT WORKCONF, MEETNOTES, ICAL, DT

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "meeting_notes."
```

```

DIM cal_default$ = "~/Documents/AutomateWork/calendar.ics"
DIM cal$ = WORKCONF.FOLDER$(cfg, key$ + "calendar", cal_default$)
DIM notes_default$ = "~/Documents/Meeting notes"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", notes_default$)
DIM days = WORKCONF.VALUE(cfg, key$ + "lookahead_days", 7)
DIM here = DT.LOCAL_OFFSET()
DIM offset = WORKCONF.VALUE(cfg, key$ + "utc_offset", here)

DIM cal = ICAL.PARSEFILE(cal$)
DIM m = MEETNOTES.NEXT_MEETING(cal, DT.NOW(), days)
IF LEN(MAP.KEYS(m)) = 0 THEN
    PRINT "No meeting in the next "; days; " days."
ELSE
    DIM file$ = MEETNOTES.FILE_NAME$(m, offset)
    DIM path$ = PATH.JOIN$(folder$, file$)
    IF FILE.EXISTS(path$) THEN
        PRINT "Already there: "; file$
    ELSEIF WORKCONF.FLAG("dry-run") THEN
        PRINT "would write "; file$
        PRINT "    in "; folder$
    ELSE
        MKDIR folder$
        MEETNOTES.WRITE_NOTES(m, path$, offset)
        PRINT "wrote "; file$
        PRINT "    in "; folder$
    ENDIF
ENDIF
ENDIF

```

The module does the work. It reads the calendar with the ICAL library, which also expands repeating meetings such as a weekly team call, and it writes the Word file with the DOCX library:

```

' meetnotes.jdb: the next meeting in a calendar file and a Word file
' to take notes in it: title, time, place, attendees, the agenda from
' the invitation, and a table for the actions.
'
'   IMPORT MEETNOTES, ICAL
'   DIM cal = ICAL.PARSEFILE("calendar.ics")
'   DIM m = MEETNOTES.NEXT_MEETING(cal, DT.NOW(), 7)
'   MEETNOTES.WRITE_NOTES(m, "notes.docx", 2)

EXPORT MODULE MEETNOTES

IMPORT ICAL, DT, DF, DOCX

' The address of an attendee value such as mailto:mia@example.com.

```

```

FUNC Address$(value$)
    IF LCASE$(LEFT$(value$, 7)) = "mailto:" THEN
        RETURN MID$(value$, 7, LEN(value$))
    ENDIF
    RETURN value$
ENDFUNC

' The agenda points of an invitation text, one per line, or three
' headings for notes when the text has none.
EXPORT FUNC AGENDA(description$)
    DIM rows = SPLIT(REPLACE$(description$, CHR$(13), ""), CHR$(10))
    DIM trimmed = SELECT(LAMBDA s$ -> TRIM$(s$), rows)
    DIM points = FILTER(LAMBDA s$ -> s$ <> "", trimmed)
    IF LEN(points) = 0 THEN
        points = ["Updates", "Decisions", "Next steps"]
    ENDIF
    RETURN points
ENDFUNC

' The first meeting that starts at now or later within the next days,
' all-day events left out. A map with "summary", "start", "end",
' "location", "attendees" and "agenda"; empty when there is none.
EXPORT FUNC NEXT_MEETING(cal, now, days)
    DIM found AS MAP
    DIM frame = ICAL.EXPAND(cal, now, now + days * 86400)
    DIM k = 0
    FOR k = 0 TO DF.NROWS(frame) - 1
        DIM r = DF.ROW(frame, k)
        DIM ev = r{"event"}
        IF r{"start"} >= now AND ALSO NOT ICAL.ALLDAY(cal, ev) THEN
            found{"summary"} = r{"summary"}
            found{"start"} = r{"start"}
            found{"end"} = r{"end"}
            found{"location"} = ICAL.PROP$(cal, ev, "LOCATION", "")
            DIM who = ICAL.VALUES(cal, ev, "ATTENDEE")
            found{"attendees"} = SELECT(Address$@, who)
            DIM desc$ = ICAL.PROP$(cal, ev, "DESCRIPTION", "")
            found{"agenda"} = AGENDA(desc$)
            RETURN found
        ENDIF
    NEXT k
    RETURN found
ENDFUNC

' A text with the characters a file name cannot hold left out.
EXPORT FUNC SAFE$(raw$)
    DIM out$ = raw$
    DIM bad = ["\\", "/", ":", "*", "?", CHR$(34), "<", ">", "|"]

```

```

    DIM k = 0
    FOR k = 0 TO LEN(bad) - 1
        out$ = REPLACE$(out$, bad[k], "")
    NEXT k
    RETURN TRIM$(out$)
ENDFUNC

' "2026-10-05 Miller project status.docx" for a meeting.
EXPORT FUNC FILE_NAME$(meeting, offset)
    DIM day$ = DT.FMT$(meeting{"start"}, "%Y-%m-%d", offset)
    RETURN day$ + " " + SAFE$(meeting{"summary"}) + ".docx"
ENDFUNC

' The Word file for a meeting; offset is the wall clock in hours east
' of UTC. Answers the number of parts written.
EXPORT FUNC WRITE_NOTES(meeting, path$, offset)
    DIM doc = DOCX.NEW()
    DOCX.LANGUAGE(doc, "en-US")
    DIM title$ = meeting{"summary"}
    DOCX.TITLE(doc, title$)
    DOCX.HEADING(doc, title$)
    DIM long_fmt$ = "%A %d %B %Y, %H:%M"
    DIM when$ = DT.FMT$(meeting{"start"}, long_fmt$, offset)
    DIM until$ = DT.FMT$(meeting{"end"}, "%H:%M", offset)
    DOCX.PARAGRAPH(doc, "**When:** " + when$ + " to " + until$)
    IF meeting{"location"} <> "" THEN
        DOCX.PARAGRAPH(doc, "**Where:** " + meeting{"location"})
    ENDIF
    DIM people = meeting{"attendees"}
    IF LEN(people) > 0 THEN
        DOCX.HEADING(doc, "Attendees", 2)
        DOCX.BULLETS(doc, people)
    ENDIF
    DOCX.HEADING(doc, "Agenda", 2)
    DOCX.NUMBERED(doc, meeting{"agenda"})
    DOCX.HEADING(doc, "Notes", 2)
    DOCX.PARAGRAPH(doc, "")
    DOCX.HEADING(doc, "Actions", 2)
    DIM rows = [{"What", "Who", "By when"}]
    DIM k = 0
    FOR k = 1 TO 4
        PUSH(rows, ["", "", ""])
    NEXT k
    DOCX.TABLE(doc, rows, TRUE)
    RETURN DOCX.WRITE(doc, path$)
ENDFUNC

```

How it works

1. ICAL.PARSEFILE reads the calendar file. ICAL.EXPAND turns it into the list of meetings in the next days, a weekly series included, in the order they start.
2. NEXT_MEETING takes the first one that has not started yet and leaves out all-day entries such as holidays, which are not meetings.
3. The attendees come from the ATTENDEE lines of the invitation:
SELECT(Address\$, who) turns every line into an address. The agenda comes from the description, one point per line: SELECT trims the lines and FILTER drops the empty ones. An invitation without a description gets three headings: Updates, Decisions, Next steps.
4. FILE_NAME\$ builds the name from the date and the title and leaves out the characters Windows does not allow in a file name, such as : and ?.
5. WRITE_NOTES writes the document: a heading, the time in your own clock, the place, the attendees as a list, the agenda as a numbered list, a space for notes and a table for the actions.

Run it

```
jdbasic meeting_notes.jdb --dry-run
would write 2026-10-05 Miller project status.docx
in C:\Users\mia\Documents\Meeting notes
```

Without --dry-run the file is written. When it is already there, the program says so and leaves your notes alone.

Schedule it

The wizard plans the Meeting Notes Starter for every weekday at 7:30. If your first meeting is often earlier, choose an earlier time on the page "Schedule".

Make it yours

The settings are in the part of work.conf that starts with [meeting_notes]:

```
[meeting_notes]
calendar = "~/Documents/AutomateWork/calendar.ics"
folder = "~/Documents/Meeting notes"
```

```
lookahead_days = 7  
utc_offset = 2
```

- **Keep notes per project:** point folder at the project folder on your team drive.
- **Look further ahead:** `lookahead_days = 14` finds the next meeting even after a holiday.
- **Your own clock:** `utc_offset` is the difference to UTC in hours. Leave it out and the program takes the one of your computer.

When it goes wrong

- **"No meeting in the next 7 days":** the calendar file is old. Export it again, or let your calendar program write it on its own.
- **The times are one hour off:** set `utc_offset` to the offset of your summer or winter time.
- **Attendees appear as mail addresses:** the recipe shows the address of each attendee, since that is what every invitation carries.

Balance dividend

About 20 minutes a week, and every meeting starts with your notes already open.

E10 Duplicate Finder

The chore

Mia's documents folder holds the same budget sheet four times: the one from the mail, the one she saved under a new name, the copy in "Old versions" and the one a colleague sent back. Photos from the site visits sit in three places. Every few months the disk is full, and finding out which copies can go takes an afternoon nobody has.

What you get

Once a week the Duplicate Finder looks through the folders you name and finds every file that is stored more than once, whatever its name. It writes an Excel report: one sheet with the totals, one with every copy, its folder and the date it last changed, the copies that waste the most space first. It deletes nothing. You decide which copy stays.

Before you start

Nothing; the wizard has entered your Documents and Downloads folders.

The program

The program the wizard runs reads the settings, collects the files, finds the groups and writes the report:

```
' Duplicate Finder: looks for files stored more than once in your
' folders and writes an Excel report of them. It deletes nothing. The
' settings are the [duplicate_finder] part of work.conf.
'
'   jdbasic duplicate_finder.jdb

IMPORT WORKCONF, DUPES

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "duplicate_finder."
DIM folders = WORKCONF.VALUE(cfg, key$ + "folders", ["~/Documents"])
DIM report_default$ = "~/Documents/AutomateWork/duplicates.xlsx"
DIM report$ = WORKCONF.FOLDER$(cfg, key$ + "report", report_default$)
DIM min_kb = WORKCONF.VALUE(cfg, key$ + "min_size_kb", 1)
```



```

DIM roots = SELECT(LAMBDA f$ -> WORKCONF.EXPAND$(f$), folders)
DIM paths = DUPES.FILES(roots)
PRINT LEN(paths); " files looked at"
DIM groups = DUPES.GROUPS(paths, min_kb * 1024)
IF LEN(groups) = 0 THEN
    PRINT "No file is stored twice."
ELSE
    DIM n_rows = DUPES.REPORT(groups, report$)
    DIM mb = ROUND(DUPES.WASTED(groups) / 1048576, 1)
    PRINT LEN(groups); " groups, "; n_rows; " files, ";
    PRINT mb; " MB in copies"
    PRINT "Report: "; report$
ENDIF

```

The module walks the folders, compares the files and writes the Excel file with the XLSX library:

```

' dupes.jdb: files stored more than once. Files of the same size are
' compared by the SHA-256 of their content; equal content makes a
' group. Nothing is deleted, the groups go into an Excel report.
'
'   IMPORT DUPES
'   DIM groups = DUPES.GROUPS(DUPES.FILES(["C:/Docs"]), 1)
'   PRINT DUPES.WASTED(groups); " bytes in copies"
'   DUPES.REPORT(groups, "duplicates.xlsx")

EXPORT MODULE DUPES

IMPORT XLSX

' Every file under a folder and its subfolders, hidden ones left out.
FUNC Walk(folder$, out)
    DIM names = DIR$(PATH.JOIN$(folder$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM full$ = PATH.JOIN$(folder$, names[k])
        DIM info = FILE.STAT(full$)
        IF NOT info{"hidden"} THEN
            IF info{"is_dir"} THEN
                Walk(full$, out)
            ELSE
                PUSH(out, full$)
            ENDIF
        ENDIF
    NEXT k
    RETURN 0
ENDFUNC

```

```

' The files under each of the folders.
EXPORT FUNC FILES(folders)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(folders) - 1
        IF FILE.ISDIR(folders[k]) THEN Walk(folders[k], out)
    NEXT k
    RETURN out
ENDFUNC

' The groups of files with the same content, each a map of "size",
' "hash" and "files"; files smaller than min_bytes are left out.
EXPORT FUNC GROUPS(paths, min_bytes)
    DIM by_size AS MAP
    DIM k = 0
    FOR k = 0 TO LEN(paths) - 1
        DIM p$ = paths[k]
        DIM sz = FILE.SIZE(p$)
        IF sz >= min_bytes AND ALSO sz > 0 THEN
            DIM skey$ = STR$(sz)
            IF NOT MAP.EXISTS(by_size, skey$) THEN by_size{skey$} = []
            DIM same_size = by_size{skey$}
            PUSH(same_size, p$)
        ENDIF
    NEXT k
    DIM out = []
    DIM sizes = MAP.KEYS(by_size)
    DIM s = 0
    FOR s = 0 TO LEN(sizes) - 1
        DIM bucket = by_size{sizes[s]}
        IF LEN(bucket) > 1 THEN out = APPEND(out, SameContent(bucket))
    NEXT s
    RETURN out
ENDFUNC

' The groups of equal content among files of one size.
FUNC SameContent(bucket)
    DIM by_hash AS MAP
    DIM order = []
    DIM k = 0
    FOR k = 0 TO LEN(bucket) - 1
        DIM p$ = bucket[k]
        DIM h$ = CODEC.SHA256$(BINREADER$(p$))
        IF NOT MAP.EXISTS(by_hash, h$) THEN
            by_hash{h$} = []
            PUSH(order, h$)
        ENDIF
    NEXT k
    RETURN order
ENDFUNC

```

```

        DIM same = by_hash{h$}
        PUSH(same, p$)
    NEXT k
    DIM out = []
    FOR k = 0 TO LEN(order) - 1
        DIM grp_files = by_hash{order[k]}
        IF LEN(grp_files) > 1 THEN
            DIM sz = FILE.SIZE(grp_files[0])
            DIM grp = {"size": sz, "hash": order[k], _
                "files": grp_files}
            PUSH(out, grp)
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' The bytes the copies take beyond one file of each group.
EXPORT FUNC WASTED(groups)
    RETURN SUM(WasteOf(groups))
ENDFUNC

' For every group the bytes its copies take.
FUNC WasteOf(groups)
    RETURN SELECT(LAMBDA g -> _
        g{"size"} * (LEN(g{"files"}) - 1), groups)
ENDFUNC

' Writes the report: a summary sheet and a sheet with a row per file,
' the largest waste first. Answers the number of file rows.
EXPORT FUNC REPORT(groups, path$)
    DIM order = SortedByWaste(groups)
    DIM rows = [{"Group", "Size (KB)", "File", "Folder", "Changed"}]
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(order) - 1
        DIM g = groups[order[k]]
        DIM grp_files = g{"files"}
        FOR j = 0 TO LEN(grp_files) - 1
            DIM f$ = grp_files[j]
            DIM kb = ROUND(g{"size"} / 1024, 1)
            DIM changed$ = FILE.STAT(f$){"mtime"}
            PUSH(rows, [k + 1, kb, PATH.BASENAME$(f$), _
                PATH.DIRNAME$(f$), changed$])
        NEXT j
    NEXT k
    DIM mb = ROUND(WASTED(groups) / 1048576, 2)
    DIM summary = [{"What", "Value"}, {"Groups", LEN(groups)}, _
        {"Files in groups", LEN(rows) - 1}, {"Wasted (MB)", mb}]

```

```

    DIM wb = XLSX.NEW()
    DIM sum_sheet = XLSX.SHEET(wb, "Summary", summary)
    XLSX.COLUMN(sum_sheet, "A", {"width": 18})
    DIM sheet = XLSX.SHEET(wb, "Duplicates", rows)
    XLSX.COLUMN(sheet, "C", {"width": 36})
    XLSX.COLUMN(sheet, "D", {"width": 48})
    XLSX.COLUMN(sheet, "E", {"width": 20})
    XLSX.FREEZE(sheet, 1)
    XLSX.WRITE(path$, wb)
    RETURN LEN(rows) - 1
ENDFUNC

' The group numbers ordered by the space their copies waste, largest
' first.
FUNC SortedByWaste(groups)
    IF LEN(groups) = 0 THEN RETURN []
    RETURN GRADE(0 - WasteOf(groups))
ENDFUNC

```

How it works

1. FILES walks every folder and its subfolders with DIR\$ and FILE.STAT and collects the paths. Hidden files stay out.
2. GROUPS sorts the files by size first. Two files of different size cannot have the same content, so only files that share a size are read at all, which keeps the program fast on a large disk.
3. SameContent reads each of those files and computes its SHA-256 with CODEC.SHA256\$, a fingerprint that is the same for equal content and different for any change, however small. Files with the same fingerprint form a group.
4. WasteOf works out for every group what its copies take beyond one file, with SELECT and a LAMBDA. WASTED is the SUM of that list.
5. REPORT writes the two sheets. GRADE orders the groups by the space they waste, so the worst offenders come first: the minus in 0 - WasteOf(groups) turns the smallest-first order around.

Run it

```
jdbasic duplicate_finder.jdb
2412 files looked at
37 groups, 81 files, 412.6 MB in copies
Report: C:\Users\mia\Documents\AutomateWork\duplicates.xlsx
```

Open the report in Excel, sort or filter as you like, and delete the copies you do not need yourself.

Schedule it

The wizard plans the Duplicate Finder for Friday at 16:00, so the report is waiting when you tidy up before the weekend.

Make it yours

The settings are in the part of `work.conf` that starts with `[duplicate_finder]`:

```
[duplicate_finder]
folders = ["~/Documents", "~/Downloads", "~/Pictures"]
report = "~/Documents/AutomateWork/duplicates.xlsx"
min_size_kb = 1
```

- **More folders:** add your Pictures folder or a project folder on the team drive to folders.
- **Only the big ones:** `min_size_kb = 1024` leaves out everything below a megabyte, which is where the space goes.

When it goes wrong

- **It takes long:** the first run on a full disk reads many files. Leave out folders such as program installations, or raise `min_size_kb`.
- **"No file is stored twice":** check the folder names in `folders`; a folder that does not exist is skipped without a message.
- **Files with the same name are missing:** the recipe compares content, not names. Two different versions of `budget.xlsx` are no duplicates.

Balance dividend

About 10 minutes a week, and the afternoon a full disk would cost.

E11 Disk Space Report

The chore

The warning comes at the worst moment: Mia is about to save the presentation for the board meeting, and Windows says the disk is full. She clicks through folders and guesses. Was it the videos from the training, the old project archives, the installers in Downloads? Ten minutes later she has freed a little space and still does not know where the rest went.

What you get

Every Monday morning the Disk Space Report measures your user folder. It lists the folders right below it with their size, the largest one first, and the ten largest files anywhere under it. Above a limit you choose it starts with a warning, long before the disk is full. The report is printed and saved as a text file in your work folder.

Before you start

Nothing; the wizard has entered your user folder.

The program

The program reads the settings, measures, prints and saves:

```
' Disk Space Report: shows where the space under a folder goes and
' warns above a limit. The report is printed and saved as a text file.
' The settings are the [disk_space] part of work.conf.
'
'   jdbasic disk_space.jdb

IMPORT WORKCONF, DISKSPACE

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "disk_space."
DIM root$ = WORKCONF.FOLDER$(cfg, key$ + "root", "~")
DIM top = WORKCONF.VALUE(cfg, key$ + "top", 10)
DIM warn_gb = WORKCONF.VALUE(cfg, key$ + "warn_gb", 50)
DIM file_default$ = "~/Documents/AutomateWork/disk_space.txt"
DIM report$ = WORKCONF.FOLDER$(cfg, key$ + "report", file_default$)
```

```

DIM scan = DISKSPACE.SCAN(root$)
DIM report_text$ = DISKSPACE.REPORT$(scan, top, warn_gb)
PRINT report_text$
MKDIR PATH.DIRNAME$(report$)
TXTWRITER report$, report_text$ + CHR$(10)
PRINT
PRINT "Saved in "; report$

```

The module walks the folders and writes the report:

```

' diskpace.jdb: where the space under a folder goes. The size of
' each folder right below it, the largest files anywhere under it,
' sizes in readable units, and a report with a warning above a limit.
'
'   IMPORT DISKSPACE
'   DIM scan = DISKSPACE.SCAN("C:/Users/mia")
'   PRINT DISKSPACE.REPORT$(scan, 10, 50)

EXPORT MODULE DISKSPACE

' Adds every file under a folder to found as [size, path] and answers
' their total size. Hidden files and folders are left out.
FUNC Walk(folder$, found)
    DIM total = 0
    DIM names = DIR$(PATH.JOIN$(folder$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM full$ = PATH.JOIN$(folder$, names[k])
        DIM info = FILE.STAT(full$)
        IF NOT info{"hidden"} THEN
            IF info{"is_dir"} THEN
                total = total + Walk(full$, found)
            ELSE
                total = total + info{"size"}
                PUSH(found, [info{"size"}, full$])
            ENDIF
        ENDIF
    NEXT k
    RETURN total
ENDFUNC

' The sizes under a folder: a map with "root", "total", "folders"
' ([name, bytes] for each folder right below it, the files directly in
' it under "(files here)" and "files" ([bytes, path] for every file).
EXPORT FUNC SCAN(root$)
    DIM found = []
    DIM folders = []

```



```

DIM loose = 0
DIM total = 0
DIM names = DIR$(PATH.JOIN$(root$, "*"))
DIM k = 0
FOR k = 0 TO LEN(names) - 1
    DIM full$ = PATH.JOIN$(root$, names[k])
    DIM info = FILE.STAT(full$)
    IF NOT info{"hidden"} THEN
        IF info{"is_dir"} THEN
            DIM sz = Walk(full$, found)
            total = total + sz
            PUSH(folders, [names[k], sz])
        ELSE
            DIM fsz = info{"size"}
            loose = loose + fsz
            total = total + fsz
            PUSH(found, [fsz, full$])
        ENDIF
    ENDIF
NEXT k
IF loose > 0 THEN PUSH(folders, ["(files here)", loose])
RETURN {"root": root$, "total": total, "folders": folders, _
        "files": found}
ENDFUNC

' The n largest entries of a list of pairs by the pair element at col.
EXPORT FUNC LARGEST(pairs, col, n)
    IF LEN(pairs) = 0 THEN RETURN []
    DIM sizes = SELECT(LAMBDA USE(col) p -> 0 - p[col], pairs)
    DIM order = TAKE(n, GRADE(sizes))
    RETURN SELECT(LAMBDA USE(pairs) i -> pairs[i], order)
ENDFUNC

' A size in bytes as "812 KB", "4.3 MB" or "1.25 GB".
EXPORT FUNC HUMAN$(bytes)
    IF bytes < 1024 THEN RETURN STR$(bytes) + " bytes"
    IF bytes < 1048576 THEN RETURN STR$(ROUND(bytes / 1024)) + " KB"
    IF bytes < 1073741824 THEN
        RETURN FORMAT$("{:.1f} MB", bytes / 1048576)
    ENDIF
    RETURN FORMAT$("{:.2f} GB", bytes / 1073741824)
ENDFUNC

' The report as text: the total with a warning when it is above
' warn_gb, the largest folders and the n largest files.
EXPORT FUNC REPORT$(scan, n, warn_gb)
    DIM out = []
    DIM total = scan{"total"}

```

```

DIM head$ = "Disk space under " + scan{"root"} + ": "
PUSH(out, head$ + HUMAN$(total))
IF total > warn_gb * 1073741824 THEN
    PUSH(out, "WARNING: more than " + STR$(warn_gb) + " GB.")
ENDIF
PUSH(out, "(Free space: see the properties of the drive.)")
PUSH(out, "")
PUSH(out, "Largest folders")
DIM big = LARGEST(scan{"folders"}, 1, n)
DIM k = 0
FOR k = 0 TO LEN(big) - 1
    DIM fsize$ = LPAD$(HUMAN$(big[k][1]), 10, " ")
    PUSH(out, fsize$ + " " + big[k][0])
NEXT k
PUSH(out, "")
PUSH(out, "Largest files")
DIM files_big = LARGEST(scan{"files"}, 0, n)
FOR k = 0 TO LEN(files_big) - 1
    DIM size$ = LPAD$(HUMAN$(files_big[k][0]), 10, " ")
    PUSH(out, size$ + " " + files_big[k][1])
NEXT k
RETURN JOIN(out, CHR$(10))
ENDFUNC

```

How it works

1. SCAN looks at each entry right below the folder. For a folder it calls walk, which adds up the sizes of every file inside it and in its subfolders. Each file is also noted with its size.
2. Hidden folders, such as the settings Windows keeps in AppData, are left out. Their size is not yours to clean up.
3. LARGEST takes the size out of every entry with SELECT, with a minus in front so the largest comes first. GRADE gives the order of those numbers, TAKE keeps the first n places, and a second SELECT picks the entries at those places. USE(col) and USE(pairs) hand each lambda a value of the function.
4. HUMAN\$ shows a size the way Explorer does: bytes, KB, MB or GB.
5. REPORT\$ puts it together and adds the warning when the total is above warn_gb.

The report does not show how much space is left on the drive, since `jdBasic` has no command for that yet on Windows. Explorer shows it in the properties of the drive.

Run it

```

jdbasic disk_space.jdb
Disk space under C:\Users\mia: 63.40 GB
WARNING: more than 50 GB.
(Free space: see the properties of the drive.)

Largest folders
38.12 GB  Videos
14.96 GB  Downloads
7.03 GB   Documents

```

The list of the largest files follows, and the last line names the text file the report was saved in.

Schedule it

The wizard plans the report for Monday at 9:00, at the start of the week, when there is still time to tidy up.

Make it yours

The settings are in the part of `work.conf` that starts with `[disk_space]`:

```

[disk_space]
root = "~"
warn_gb = 50
top = 10
report = "~/Documents/AutomateWork/disk_space.txt"

```

- **A team drive:** set `root` to the folder of your team, such as `"S:/Projects"`, to see which project grows fastest.
- **A smaller disk:** lower `warn_gb` to half the size of your disk.
- **More detail:** `top = 25` lists more folders and files.

When it goes wrong

- **It takes a while:** measuring a large folder means looking at every file in it once. That is why the recipe runs on Monday morning on its own.
- **A folder is missing from the list:** it is hidden, or you have no right to read it.
- **The sizes differ from Explorer:** Explorer counts the space a file takes on the disk, the report counts its length. Small files take a little more room than they are long.

Balance dividend

About 5 minutes a week, and no full disk on the morning it matters.

E12 Birthday Reminder

The chore

Mia keeps the birthdays of her team, a few clients and her family in a spreadsheet. She means to look at it every Monday. Most weeks she forgets, and twice this year she noticed a birthday when the cake was already on the table. Typing forty birthdays into the calendar by hand is a job she has put off for two years.

What you get

Every morning the Birthday Reminder reads your contact list and prints the birthdays and anniversaries of the next two weeks, the nearest first, with the age the person turns. It also writes a calendar file with a yearly event for every date on the list. Import it once into Outlook or Google Calendar, or subscribe to it, and the dates appear in your calendar every year without further typing.

Before you start

Save your contacts as `contacts.csv` in your work folder, with a header row and one person per line. Excel saves this format with *Save As, CSV (Comma delimited)*:

```
name,birthday,anniversary
Ann Miller,1985-10-03,2012-06-15
Larry Leap,1992-02-29,
Jan Newyear,1979-01-02,
```

Write the dates as year, month and day with dashes. Leave the anniversary empty when there is none.

The program

The program reads the settings, prints the list and writes the calendar file:

```
' Birthday Reminder: lists the birthdays and anniversaries of your
' contacts in the next days and keeps a calendar file with all of them
' up to date. The settings are the [birthday_reminder] part of
' work.conf.
'
```

```

'   jdbcbasic birthday_reminder.jdb          list, write the calendar
'   jdbcbasic birthday_reminder.jdb --dry-run  only list

IMPORT WORKCONF, BIRTHDAYS, DT

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "birthday_reminder."
DIM home$ = "~/Documents/AutomateWork/"
DIM csv_default$ = home$ + "contacts.csv"
DIM csv$ = WORKCONF.FOLDER$(cfg, key$ + "contacts", csv_default$)
DIM ics_default$ = home$ + "birthdays.ics"
DIM ics$ = WORKCONF.FOLDER$(cfg, key$ + "calendar", ics_default$)
DIM horizon = WORKCONF.VALUE(cfg, key$ + "days", 14)

DIM today$ = DT.FMT$(DT.NOW(), "%Y-%m-%d", DT.LOCAL_OFFSET())
DIM people = BIRTHDAYS.CONTACTS(csv$)
DIM soon = BIRTHDAYS.UPCOMING(people, today$, horizon)
IF LEN(soon) = 0 THEN
    PRINT "No birthday in the next "; horizon; " days."
ELSE
    DIM k = 0
    FOR k = 0 TO LEN(soon) - 1
        PRINT BIRTHDAYS.ENTRY$(soon[k])
    NEXT k
ENDIF
IF NOT WORKCONF.FLAG("dry-run") THEN
    DIM n = BIRTHDAYS.CALENDAR(people, today$, ics$)
    PRINT n; " yearly events in "; ics$
ENDIF

```

The module reads the contacts, finds the next date of each birthday and anniversary, and builds the calendar with the ICAL library:

```

' birthdays.jdb: the birthdays and anniversaries of a contact list in
' the next days, with the age or the years, and a calendar file with a
' yearly event for each. A birthday on 29 February is kept on the 28th
' in other years.
'
'   IMPORT BIRTHDAYS
'   DIM people = BIRTHDAYS.CONTACTS("contacts.csv")
'   DIM soon = BIRTHDAYS.UPCOMING(people, "2026-10-01", 14)
'   BIRTHDAYS.CALENDAR(people, "2026-10-01", "birthdays.ics")

EXPORT MODULE BIRTHDAYS

IMPORT DT, ICAL

```

```

' The contacts of a CSV file with the columns name, birthday and
' anniversary (dates as YYYY-MM-DD, an empty cell for none).
EXPORT FUNC CONTACTS(path$)
    DIM kinds = ["STRING", "STRING", "STRING"]
    DIM rows = CSVREADER(path$, ",", TRUE, kinds)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM r = rows[k]
        DIM anniv$ = ""
        IF LEN(r) > 2 THEN anniv$ = TRIM$(r[2])
        DIM who$ = TRIM$(r[0])
        DIM born$ = TRIM$(r[1])
        PUSH(out, {"name": who$, "birthday": born$, _
            "anniversary": anniv$})
    NEXT k
    RETURN out
ENDFUNC

FUNC Part(date$, from, n)
    RETURN CINT(MID$(date$, from, n))
ENDFUNC

' The day of a yearly date in a given year; 29 February moves to the
' 28th in common years.
FUNC DayIn(yr, date$)
    DIM mo = Part(date$, 5, 2)
    DIM dd = Part(date$, 8, 2)
    IF mo = 2 AND ALSO dd = 29 AND ALSO NOT DT.IS_LEAP(yr) THEN dd = 28
    RETURN DT.MAKE(yr, mo, dd)
ENDFUNC

' The next day on or after today$ that a yearly date comes back.
EXPORT FUNC NEXT_DATE(date$, today$)
    DIM today = DT.PARSE(today$)
    DIM yr = Part(today$, 0, 4)
    DIM on_day = DayIn(yr, date$)
    IF on_day < today THEN on_day = DayIn(yr + 1, date$)
    RETURN on_day
ENDFUNC

SUB AddEvent(out, who$, what$, date$, today$, horizon)
    IF date$ = "" THEN RETURN
    DIM on_day = NEXT_DATE(date$, today$)
    DIM n_days = DT.DIFF(DT.PARSE(today$), on_day, "days")
    IF n_days > horizon THEN RETURN
    DIM when$ = DT.FMT$(on_day, "%Y-%m-%d")
    DIM years = Part(when$, 0, 4) - Part(date$, 0, 4)

```

```

    PUSH(out, {"date": when$, "in_days": n_days, "name": who$, _
              "what": what$, "years": years})
ENDSUB

' The birthdays and anniversaries from today$ to horizon days later,
' the nearest first: maps with "date", "in_days", "name", "what" and
' "years".
EXPORT FUNC UPCOMING(people, today$, horizon)
    DIM found = []
    DIM k = 0
    FOR k = 0 TO LEN(people) - 1
        DIM p = people[k]
        AddEvent(found, p{"name"}, "birthday", p{"birthday"}, _
                 today$, horizon)
        AddEvent(found, p{"name"}, "anniversary", p{"anniversary"}, _
                 today$, horizon)
    NEXT k
    IF LEN(found) = 0 THEN RETURN found
    DIM waits = SELECT(LAMBDA e -> e{"in_days"}, found)
    DIM order = GRADE(waits)
    RETURN SELECT(LAMBDA USE(found) i -> found[i], order)
ENDFUNC

' One line for an entry, such as
' "Sat 03 Oct Ann Miller turns 41 (in 2 days)".
EXPORT FUNC ENTRY$(e)
    DIM when$ = DT.FMT$(DT.PARSE(e{"date"}), "%a %d %b")
    DIM n = e{"in_days"}
    DIM soon$ = "today"
    IF n = 1 THEN soon$ = "tomorrow"
    IF n > 1 THEN soon$ = "in " + STR$(n) + " days"
    DIM what$ = e{"name"} + " turns " + STR$(e{"years"})
    IF e{"what"} = "anniversary" THEN
        what$ = e{"name"} + ": " + STR$(e{"years"}) + " years"
    ENDIF
    RETURN when$ + " " + what$ + " (" + soon$ + ")"
ENDFUNC

FUNC AddYearly(cal, title$, date$, today$)
    IF date$ = "" THEN RETURN 0
    DIM ev = ICAL.ADDALLDAY(cal, title$, NEXT_DATE(date$, today$))
    DIM rule$ = "FREQ=YEARLY"
    IF MID$(date$, 5, 5) = "02-29" THEN
        rule$ = "FREQ=YEARLY;BYMONTH=2;BYMONTHDAY=-1"
    ENDIF
    ICAL.SETPROP(cal, ev, "RRULE", rule$)
    RETURN 1
ENDFUNC

```



```
' A calendar file with a yearly all-day event for every birthday and
' anniversary, starting from its next date. Answers the event count.
EXPORT FUNC CALENDAR(people, today$, path$)
    DIM cal = ICAL.NEW()
    DIM n = 0
    DIM k = 0
    FOR k = 0 TO LEN(people) - 1
        DIM p = people[k]
        DIM who$ = p{"name"}
        n = n + AddYearly(cal, "Birthday: " + who$, p{"birthday"}, _
            today$)
        n = n + AddYearly(cal, "Anniversary: " + who$, _
            p{"anniversary"}, today$)
    NEXT k
    ICAL.WRITEFILE(cal, path$)
    RETURN n
ENDFUNC
```

How it works

1. CONTACTS reads the CSV file with CSVREADER. All three columns are read as text, so a date such as 1985-10-03 stays exactly as written.
2. NEXT_DATE puts the day and month of a date into this year. If that day has passed, it takes next year. A birthday on 29 February falls on the 28th in years that have no 29th.
3. UPCOMING keeps the dates within the next days days. SELECT takes out the days to wait, GRADE gives their order, the nearest first, and a second SELECT picks the entries in that order; USE(found) hands that lambda the list of the function. The age is the year of the next date minus the year of birth.
4. ENTRY\$ writes one line for each, such as "Sat 03 Oct Ann Miller turns 41 (in 2 days)".
5. CALENDAR writes an all-day event for each date that repeats every year. For 29 February the rule says "the last day of February", which is the 29th in leap years and the 28th in all others.

Run it

```
jdbasic birthday_reminder.jdb
Sat 03 Oct Ann Miller turns 41 (in 2 days)
Mon 12 Oct Tom Berger turns 30 (in 11 days)
4 yearly events in C:\Users\mia\Documents\AutomateWork\birthdays.ics
```

With `--dry-run` only the list is printed.

Schedule it

The wizard plans the reminder for 8:00 every morning, so the list is the first thing you see when the day starts.

Make it yours

The settings are in the part of `work.conf` that starts with `[birthday_reminder]`:

```
[birthday_reminder]
contacts = "~/Documents/AutomateWork/contacts.csv"
calendar = "~/Documents/AutomateWork/birthdays.ics"
days = 14
```

- **More notice for gifts:** `days = 30` shows a month ahead.
- **One list for the team:** put `contacts.csv` on the team drive and point `contacts` at it, so everyone works from the same list.

When it goes wrong

- **A person is missing:** check the date in the CSV file. It has to be written as `1985-10-03`; `03.10.1985` is not read as a date.
- **The age is odd:** the year of birth is unknown and was entered as the current year. Leave the year as it is and ignore the age, or enter the real one.
- **The calendar shows every date twice:** the file was imported twice. Subscribe to it instead of importing it, or delete the first import.

Balance dividend

About 10 minutes a week, and no forgotten birthday at the office.

Before you start

The setup wizard has run and the Printable Week is switched on. Your to-do list is a plain text file, by default `todo.txt` in your work folder, with one task per line:

```
Mon: Call Miller about the offer
Tue: ! Send invoice RE-0413
Wed: Order toner
Plan the team day
# a line starting with # is a note to yourself
```

A day before the colon puts the task on that day; Mon, monday and Monday all work. A ! marks a task important. A line without a day belongs to the whole week.

The program

The program reads the settings, finds the Monday of the week, asks the planner for the plan and writes the page:

```
' Printable Week: turns the to-do list into a one-page plan of the
' week, a column per day. The settings are the [printable_week] part
' of work.conf.
'
'   jdbcbasic printable_week.jdb           write the page
'   jdbcbasic printable_week.jdb --dry-run  show the plan only

IMPORT WORKCONF, PLANNER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "printable_week."
DIM home$ = "~/Documents/AutomateWork"
DIM todo_default$ = home$ + "/todo.txt"
DIM todo$ = WORKCONF.FOLDER$(cfg, key$ + "todo_file", todo_default$)
DIM out_default$ = home$ + "/print"
DIM out$ = WORKCONF.FOLDER$(cfg, key$ + "out_folder", out_default$)
DIM n_days = WORKCONF.VALUE(cfg, key$ + "days", 5)
DIM ahead = WORKCONF.VALUE(cfg, key$ + "weeks_ahead", 1)

IF NOT FILE.EXISTS(todo$) THEN THROW "No to-do list at " + todo$
DIM tasks = PLANNER.PARSE(TXTREADER$(todo$))
DIM monday = PLANNER.MONDAY(DATEADD("D", 7 * ahead, NOW()))
DIM plan = PLANNER.PLAN(tasks, monday, n_days)
PRINT plan{"title"}
```

```

DIM columns = plan{"columns"}
DIM k = 0
FOR k = 0 TO LEN(columns) - 1
    DIM col = columns[k]
    DIM n_tasks = LEN(col{"tasks"})
    DIM word$ = " tasks"
    IF n_tasks = 1 THEN word$ = " task"
    PRINT " "; col{"label"}; ": "; n_tasks; word$
NEXT k
IF NOT WORKCONF.FLAG("dry-run") THEN
    MKDIR out$
    DIM week$ = LEFT$(plan{"title"}, INSTR(plan{"title"}, ":"))
    DIM pdf$ = PATH.JOIN$(out$, week$ + ".pdf")
    DIM result = PLANNER.WRITE(plan, pdf$)
    PRINT "Wrote "; pdf$
    IF result{"hidden"} > 0 THEN
        PRINT result{"hidden"}; " tasks did not fit; see + more."
    ENDIF
ENDIF
ENDIF

```

The planner reads the list, sorts the tasks into columns and draws the page with PDFGEN:

```

' planner.jdb: a plain to-do list turned into a one-page week plan
' for the desk. A line "Mon: Call Miller" belongs to Monday, a line
' without a day to the week as a whole, a "!" marks it important.
'
'   IMPORT PLANNER
'   DIM tasks = PLANNER.PARSE(TXTREADER$("todo.txt"))
'   DIM plan = PLANNER.PLAN(tasks, PLANNER.MONDAY(NOW()), 5)
'   PRINT PLANNER.WRITE(plan, "week.pdf"){"placed"}

EXPORT MODULE PLANNER

IMPORT PDFGEN

DIM gDays = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

' The day a word names, "Mon" to "Sun", or "" when it names none.
FUNC DayOf$(word$)
    DIM w$ = LCASE$(TRIM$(word$))
    IF LEN(w$) < 3 ORELSE LEN(w$) > 9 THEN RETURN ""
    DIM k = 0
    FOR k = 0 TO 6
        DIM d$ = gDays[k]
        IF LEFT$(w$, 3) = LCASE$(d$) THEN RETURN d$
    NEXT k

```

```

RETURN ""
ENDFUNC

' The tasks of a to-do text: a map per line with "day" ("" for the
' whole week), "text" and "important". Empty lines and lines starting
' with # are skipped.
EXPORT FUNC PARSE(text$)
    DIM tasks = []
    DIM rows = SPLIT(REPLACE$(text$, CHR$(13), ""), CHR$(10))
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM t$ = TRIM$(rows[k])
        IF t$ <> "" AND ALSO NOT STARTSWITH(t$, "#") THEN
            DIM day$ = ""
            DIM colon = CINT(INSTR(t$, ":"))
            IF colon > 0 THEN day$ = DayOf$(LEFT$(t$, colon))
            DIM rest$ = MID$(t$, colon + 1, LEN(t$))
            IF day$ <> "" THEN t$ = TRIM$(rest$)
            DIM important = STARTSWITH(t$, "!")
            IF important THEN t$ = TRIM$(MID$(t$, 1, LEN(t$)))
            IF t$ <> "" THEN
                PUSH(tasks, {"day": day$, "text": t$, _
                    "important": important})
            ENDIF
        ENDIF
    NEXT k
    RETURN tasks
ENDFUNC

' The Monday of the week a date falls in. The weekday comes from
' FORMAT_DATE, 0 for Sunday.
EXPORT FUNC MONDAY(d)
    DIM wd = VAL(FORMAT_DATE(d, "%w"))
    RETURN DATEADD("D", -((wd + 6) MOD 7), d)
ENDFUNC

' The week as columns: one per day (n_days 5 or 7) and a last one for
' the tasks of the whole week, which also takes the weekend's tasks
' when the plan has five days. Answers "title" and "columns", each
' column a map of "label" and "tasks".
EXPORT FUNC PLAN(tasks, monday, n_days)
    DIM columns = []
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO n_days - 1
        DIM d = DATEADD("D", k, monday)
        DIM label$ = gDays[k] + " " + STR$(DAY(d))
        PUSH(columns, {"label": label$, "tasks": []})
    NEXT k
    DIM title$ = "Tasks of the week"
    PUSH(columns, {"label": title$, "tasks": tasks})
    RETURN columns

```

```

NEXT k
PUSH(columns, {"label": "This week", "tasks": []})
FOR j = 0 TO LEN(tasks) - 1
    DIM t = tasks[j]
    DIM at = n_days
    FOR k = 0 TO n_days - 1
        IF t{"day"} = gDays[k] THEN at = k
    NEXT k
    DIM col = columns[at]
    DIM bucket = col{"tasks"}
    PUSH(bucket, t)
NEXT j
RETURN {"title": TITLE$(monday, n_days), "columns": columns}
ENDFUNC

' "Week 41: 5 to 9 October 2026", with both months when it spans two.
EXPORT FUNC TITLE$(monday, n_days)
    DIM last = DATEADD("D", n_days - 1, monday)
    DIM first$ = STR$(DAY(monday))
    IF MONTH(monday) <> MONTH(last) THEN
        first$ = first$ + " " + FORMAT_DATE(monday, "%B")
    ENDIF
    DIM week$ = "Week " + STR$(VAL(FORMAT_DATE(monday, "%V")))
    DIM span$ = first$ + " to " + STR$(DAY(last)) + " "
    RETURN week$ + ": " + span$ + FORMAT_DATE(last, "%B %Y")
ENDFUNC

' Writes the plan as one A4 page, landscape: a column per day with a
' box to tick before every task and lines to write on below. Answers
' "placed" (tasks on the page) and "hidden" (tasks that did not fit,
' counted as "+ n more").
EXPORT FUNC WRITE(plan, path$)
    DIM pdf = PDFGEN.DOC({"size": "A4", "landscape": TRUE, _
        "margin": 12})
    PDFGEN.SETINFO(pdf, plan{"title"}, "", "Printable Week")
    PDFGEN.ADDPAGE(pdf)
    PDFGEN.USEFONT(pdf, "helvetica", "B", 16)
    PDFGEN.TEXTAT(pdf, 12, 20, plan{"title"})
    DIM columns = plan{"columns"}
    DIM n = LEN(columns)
    DIM col_w = (297 - 24) / n
    DIM bottom = 210 - 14
    DIM placed = 0
    DIM hidden = 0
    DIM k = 0
    FOR k = 0 TO n - 1
        DIM col = columns[k]
        DIM x = 12 + k * col_w

```

```

    PDFGEN.SETFILL(pdf, 225, 233, 245)
    PDFGEN.BOX(pdf, x, 26, col_w - 2, 8, "F")
    PDFGEN.USEFONT(pdf, "helvetica", "B", 10)
    PDFGEN.SETCOLOR(pdf, 31, 56, 100)
    PDFGEN.TEXTAT(pdf, x + 2, 31.5, col{"label"})
    PDFGEN.SETCOLOR(pdf, 0, 0, 0)
    DIM col_tasks = col{"tasks"}
    DIM shown = Column(pdf, col_tasks, x, 38, col_w - 2, bottom)
    placed = placed + shown
    hidden = hidden + LEN(col_tasks) - shown
NEXT k
PDFGEN.WRITEFILE(pdf, path$)
RETURN {"placed": placed, "hidden": hidden}
ENDFUNC

' The tasks of one column from y down to bottom; answers how many fit.
FUNC Column(pdf, tasks, x, y, w, bottom)
    DIM lead = 4.6
    DIM k = 0
    FOR k = 0 TO LEN(tasks) - 1
        DIM t = tasks[k]
        DIM sty$ = ""
        IF t{"important"} THEN sty$ = "B"
        PDFGEN.USEFONT(pdf, "helvetica", sty$, 9.5)
        DIM rows = PDFGEN.SPLITLINES(pdf, t{"text"}, w - 7)
        DIM need = LEN(rows) * lead + 2
        IF y + need > bottom - lead THEN
            PDFGEN.USEFONT(pdf, "helvetica", "I", 9)
            DIM more$ = "+ " + STR$(LEN(tasks) - k) + " more"
            PDFGEN.TEXTAT(pdf, x + 1, y + 3.5, more$)
            RETURN k
        ENDIF
        PDFGEN.SETDRAW(pdf, 120, 120, 120)
        PDFGEN.SETLINEWIDTH(pdf, 0.3)
        PDFGEN.BOX(pdf, x + 1, y + 0.6, 3.4, 3.4, "D")
        DIM j = 0
        FOR j = 0 TO LEN(rows) - 1
            PDFGEN.TEXTAT(pdf, x + 6.5, y + 3.5 + j * lead, rows[j])
        NEXT j
        y = y + need
    NEXT k
    Ruled(pdf, x, y, w, bottom)
    RETURN LEN(tasks)
ENDFUNC

' Light lines below y down to bottom, for tasks written in by hand,
' on a 7 mm grid that is the same in every column.
SUB Ruled(pdf, x, y, w, bottom)

```



```

PDFGEN.SETDRAW(pdf, 215, 215, 215)
PDFGEN.SETLINEWIDTH(pdf, 0.2)
DIM ly = 38
DO WHILE ly < y + 3
    ly = ly + 7
LOOP
DO WHILE ly <= bottom
    PDFGEN.DRAWLINE(pdf, x + 1, ly, x + w - 1, ly)
    ly = ly + 7
LOOP
ENDSUB

```

How it works

1. `PLANNER.PARSE` reads the list line by line. When the word before the first colon names a day, the task belongs to that day; anything else, such as `Note: call back`, keeps its colon and goes to the week.
2. `PLANNER.MONDAY` finds the Monday of the week a date falls in. The program asks for the week `weeks_ahead` weeks from today, so a run on Friday plans the next week.
3. `PLANNER.PLAN` makes a column per day and one for the week. With five days, tasks for Saturday and Sunday go to the week's column.
4. `PLANNER.WRITE` draws the page: a shaded heading per column, a box and the wrapped text for every task, and ruled lines below. A column that runs out of room ends with a line such as `+ 3 more`, so no task disappears without a trace.

Run it

With `--dry-run` the program shows the plan and writes nothing:

```

jdbasic printable_week.jdb --dry-run
Week 41: 5 to 9 October 2026
  Mon 5: 1 task
  Tue 6: 1 task
  Wed 7: 1 task
  Thu 8: 1 task
  Fri 9: 0 tasks
This week: 2 tasks

```

Without the switch it also writes `Week 41.pdf` into the folder for printable pages. Open it and print it.

Schedule it

The wizard plans the Printable Week for every Friday at 16:00, so the page for the next week is waiting when you pack up. Change the time on the wizard's page "Schedule".

Make it yours

The settings are in the `[printable_week]` part of `work.conf`:

```
[printable_week]
todo_file = "~/Documents/AutomateWork/todo.txt"
out_folder = "~/Documents/AutomateWork/print"
days = 5
weeks_ahead = 1
```

- **Plan the weekend too:** `days = 7` adds Saturday and Sunday.
- **Print this week instead of the next:** `weeks_ahead = 0`, useful when the program runs on Monday morning.
- **Keep the list elsewhere:** point `todo_file` to the file you already use, for example in a synced folder.

When it goes wrong

- **"No to-do list":** `todo_file` names a file that does not exist. Create it or correct the path.
- **A task is on the wrong day:** check the word before the colon. Only the first three letters count, so `Mo:` is not Monday.
- **"tasks did not fit":** a day holds more than the column takes. Move some tasks to the week or split the day.

Balance dividend

About 10 minutes every Friday, and a Monday that starts with the week already on paper.

E14 Page Watcher

The chore

Mia's team needs a new desk lamp, and the one they like is too expensive. The shop lowers its prices now and then, so Mia opens the page every morning to look. She does the same for the opening hours of the town hall, which change in the holidays, and for the status page of the supplier's ordering system. Five pages, two minutes each, every day, and on most days nothing has changed.

What you get

Every morning the Page Watcher fetches the pages you named, cuts out the part you care about, a price, a date, a status line, and compares it with what it saw the time before. It tells you only what changed, word by word:

```
Desk lamp: changed
    [-19.99-]{+17.49+} EUR
Town hall hours: same
Supplier status: same
```

Before you start

The setup wizard has run and the Page Watcher is switched on. For every page you want to watch you need two things: its address, and two short pieces of the page's HTML around the part you care about. Open the page in your browser, right-click the price and choose "View page source" or "Inspect". Around the price you will find something like `<span id=price>19.99 EUR</span>`. The text just before the price, `id=price>`, is the start marker; `</span>` after it is the end marker.

The program

The program reads the list of pages and checks them one after the other:

```
' Page Watcher: looks at the web pages listed in work.conf and tells
' you which of them changed in the part you watch. The settings are
' the [page_watcher] part of work.conf, one [[page_watcher.page]] per
' page.
'
```

```

'   jdbasic page_watcher.jdb                check and remember
'   jdbasic page_watcher.jdb --dry-run      check, remember nothing

IMPORT WORKCONF, WATCHER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM home$ = "~/Documents/AutomateWork/watch"
DIM store$ = WORKCONF.FOLDER$(cfg, "page_watcher.store", home$)
DIM pages = WORKCONF.VALUE(cfg, "page_watcher.page", [])
DIM dry_run = WORKCONF.FLAG("dry-run")
IF LEN(pages) = 0 THEN PRINT "No pages to watch in work.conf."

DIM changes = []
DIM k = 0
FOR k = 0 TO LEN(pages) - 1
    DIM page = pages[k]
    DIM r = WATCHER.CHECK(page, store$, dry_run)
    DIM state$ = r{"status"}
    PRINT page{"name"}; ": "; state$
    IF state$ = "changed" ORELSE state$ = "error" THEN
        PRINT "      "; r{"diff"}
    ENDIF
    IF state$ = "changed" THEN
        PUSH(changes, page{"name"} + ": " + r{"diff"})
    ENDIF
NEXT k
IF LEN(changes) > 0 AND ALSO NOT dry_run THEN
    DIM stamp$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M")
    DIM entry$ = stamp$ + CHR$(10) + JOIN(changes, CHR$(10))
    DIM log_file$ = PATH.JOIN$(store$, "changes.log")
    TXTWRITER log_file$, entry$ + CHR$(10), TRUE
ENDIF

```

The watcher fetches a page, takes the part between the markers, turns it into plain text and compares it with the stored copy:

```

' watcher.jdb: tells whether the part of a web page you care about
' differs from the last look. The part lies between two markers in
' the page's HTML; its text is kept in a folder and compared word by
' word with the copy from the time before.
'
'   IMPORT WATCHER
'   DIM page = {"name": "Desk lamp", "url": url$, _
'       "start": "id=price>", "end": "</span>"}
'   DIM r = WATCHER.CHECK(page, "C:/watch", FALSE)
'   IF r{"status"} = "changed" THEN PRINT r{"diff"}

```

```
EXPORT MODULE WATCHER
```

```
IMPORT TEXTDIFF
```

```
' The readable text of HTML: scripts, styles and tags removed, the
' common entities turned back into characters, spaces collapsed.
```

```
EXPORT FUNC PLAINTEXT$(html$)
```

```
    DIM t$ = REGEX.REPLACE("<script[\s\S]*?</script>", html$, " ")
    t$ = REGEX.REPLACE("<style[\s\S]*?</style>", t$, " ")
    t$ = REGEX.REPLACE("<[^>]*>", t$, " ")
    t$ = REPLACE$(REPLACE$(t$, "&nbsp;", " "), "&quot;", CHR$(34))
    t$ = REPLACE$(REPLACE$(t$, "&#39;", "'"), "&lt;", "<")
    t$ = REPLACE$(REPLACE$(t$, "&gt;", ">"), "&amp;", "&")
    RETURN TRIM$(REGEX.REPLACE("\s+", t$, " "))
```

```
ENDFUNC
```

```
' The text between the first start marker and the end marker after
' it; an empty marker means the start or the end of the page. A start
' marker the page does not contain gives "".
```

```
EXPORT FUNC PART$(html$, start$, end$)
```

```
    DIM part$ = html$
    IF start$ <> "" THEN
        DIM at = CINT(INSTR(part$, start$))
        IF at < 0 THEN RETURN ""
        part$ = MID$(part$, at + LEN(start$), LEN(part$))
    ENDIF
    IF end$ <> "" THEN
        DIM stop_at = CINT(INSTR(part$, end$))
        IF stop_at >= 0 THEN part$ = LEFT$(part$, stop_at)
    ENDIF
    RETURN PLAINTEXT$(part$)
```

```
ENDFUNC
```

```
' A file name made from a page's name: letters and digits, the rest
' as underscores.
```

```
EXPORT FUNC SLUG$(name$)
```

```
    DIM s$ = REGEX.REPLACE("[^a-z0-9]+", LCASE$(name$), "_")
    IF s$ = "" OR ELSE s$ = "_" THEN s$ = "page"
    RETURN s$
```

```
ENDFUNC
```

```
' The page at a URL: "ok", "body" and, when it failed, "error".
```

```
EXPORT FUNC FETCH(url$)
```

```
    DIM result AS MAP
    result{"ok"} = FALSE
    result{"body"} = ""
    result{"error"} = ""
    TRY
```

```

    DIM body$ = HTTP.GET$(url$)
    DIM code = HTTP.STATUSCODE()
    IF code >= 200 AND ALSO code < 300 THEN
        result{"ok"} = TRUE
        result{"body"} = body$
    ELSE
        result{"error"} = "the server answered " + STR$(code)
    ENDIF
CATCH
    result{"error"} = "cannot reach " + url$
ENDTRY
RETURN result
ENDFUNC

' The watched text compared with the copy in store$: "status" is
' "new", "same" or "changed", "diff" marks the changed words as
' [-old-]{+new+}. The new text is kept unless dry_run is set.
EXPORT FUNC COMPARE(name$, text$, store$, dry_run)
    DIM file$ = PATH.JOIN$(store$, SLUG$(name$) + ".txt")
    DIM out AS MAP
    out{"diff"} = ""
    out{"before"} = ""
    IF NOT FILE.EXISTS(file$) THEN
        out{"status"} = "new"
    ELSE
        DIM before$ = TXTREADER$(file$)
        out{"before"} = before$
        IF before$ = text$ THEN
            out{"status"} = "same"
        ELSE
            out{"status"} = "changed"
            out{"diff"} = TEXTDIFF.WORDDIFF$(before$, text$)
        ENDIF
    ENDIF
    IF NOT dry_run AND ALSO out{"status"} <> "same" THEN
        MKDIR store$
        TXTWRITER file$, text$
    ENDIF
    RETURN out
ENDFUNC

' One page from fetching to comparing; "status" is also "error" when
' the page cannot be read or "missing" when its start marker is gone.
EXPORT FUNC CHECK(page, store$, dry_run)
    DIM got = FETCH(page{"url"})
    IF NOT got{"ok"} THEN
        RETURN {"status": "error", "diff": got{"error"}}
    ENDIF

```

```

DIM start$ = ""
DIM end$ = ""
IF MAP.EXISTS(page, "start") THEN start$ = page{"start"}
IF MAP.EXISTS(page, "end") THEN end$ = page{"end"}
DIM text$ = PART$(got{"body"}, start$, end$)
IF text$ = "" THEN
    RETURN {"status": "missing", "diff": "marker not found"}
ENDIF
RETURN COMPARE(page{"name"}, text$, store$, dry_run)
ENDFUNC

```

How it works

1. WATCHER.FETCH asks for the page with HTTP.GET\$. A server that does not answer or answers with an error code gives a message instead of a page, and the program goes on with the next page.
2. WATCHER.PART\$ finds the start marker, takes what follows up to the end marker and hands it to PLAINTEXT\$, which removes the HTML tags, scripts and styles and turns entities such as & back into characters. What is left is the text a reader sees.
3. WATCHER.COMPARE keeps one text file per page in the store folder. The first look stores the text and reports new. Later looks report same or changed; for a change TEXTDIFF.WORDDIFF\$ marks the words that went as [-old-] and the words that came as [+new+].
4. Every change is also added to changes.log in the store folder, with the date, so you can see later when the price dropped.

Run it

The first run stores what it finds:

```

jdbasic page_watcher.jdb
Desk lamp: new

```

Later runs report same until the part changes. With `--dry-run` the program checks and reports but remembers nothing, which is the way to test new markers.

Schedule it

The wizard plans the Page Watcher for every morning at 8:00. Pages that change more often can be checked every hour; choose "every 1 hours" on the wizard's page "Schedule".

Make it yours

The store folder is in the [page_watcher] part of work.conf. Each page is a [[page_watcher.page]] block of its own, which you add by hand:

```
[page_watcher]
store = "~/Documents/AutomateWork/watch"

[[page_watcher.page]]
name = "Desk lamp"
url = "https://shop.example.com/lamps/4711"
start = "id=price>"
end = "</span>"

[[page_watcher.page]]
name = "Town hall hours"
url = "https://www.example-town.de/opening-hours"
start = "<h2>Opening hours</h2>"
end = "</table>"
```

- **Watch a whole page:** leave out start and end. Small changes such as a date in the footer then count too.
- **Watch more pages:** copy a block and change its name, address and markers. The name also names the stored copy, so keep it unique.

When it goes wrong

- **missing:** the start marker is no longer on the page. Shops change their pages; look at the source again and update the markers.
- **error with "cannot reach":** the address is wrong or you are offline. The program tries again on its next run.
- **changed every day:** the part holds something that always changes, such as a time or a visitor counter. Narrow it with closer markers.
- **A shop shows a page that asks you to log in:** some pages need a browser session. Those pages are for the expert recipe X11.

Balance dividend

About 20 minutes a week of opening the same pages, and you hear about the lower price on the morning it drops.

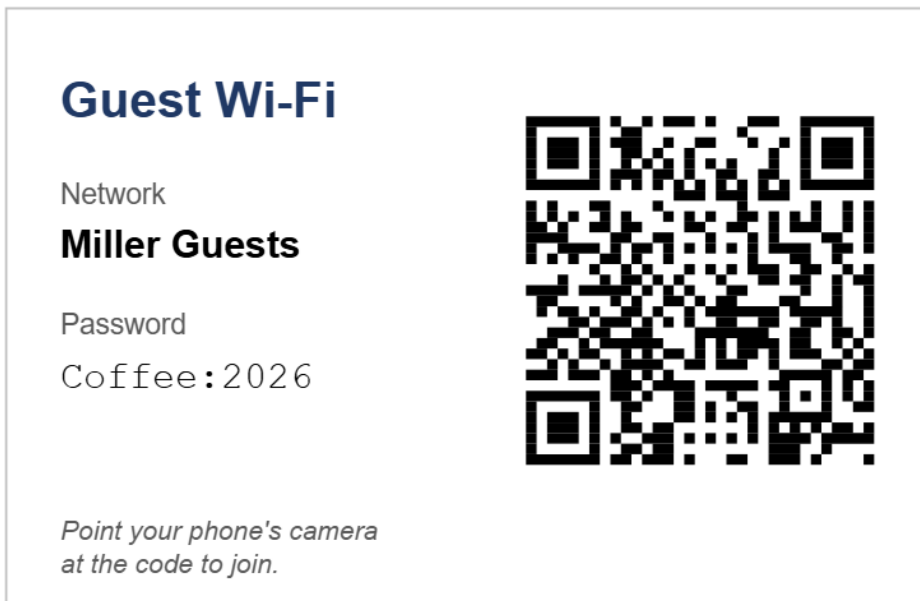
E15 Guest Wi-Fi Card

The chore

Visitors come to Mia's office every day, and almost every one of them asks for the Wi-Fi. Mia spells out the network name, then the password, twice, because "the second letter is a capital and the zero is a zero". The card she once wrote by hand has gone missing from the meeting room.

What you get

A page of eight cards to cut out and put on the tables in the meeting rooms. Each card shows the network's name and password, and a QR code. A visitor points the phone's camera at the code and the phone offers to join the network; nobody has to type anything.



A guest Wi-Fi card with its QR code

Before you start

The setup wizard has run and asked for the name and the password of the guest network. If your office has no guest network, ask whoever runs the Wi-Fi to set one up: a guest network keeps visitors away from the computers and printers of the office, and its password can be printed without worry.

The program

The program reads the network's details, builds the text of the QR code and writes the page of cards:

```
' Guest Wi-Fi Card: prints cards with the guest network's name,
' password and a QR code that joins a phone to it. The settings are
' the [guest_wifi] part of work.conf.
'
' jdbasic guest_wifi.jdb          write the page of cards
' jdbasic guest_wifi.jdb --dry-run show the code in the terminal

IMPORT WORKCONF, WIFICARD, QR

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "guest_wifi."
DIM ssid$ = WORKCONF.VALUE(cfg, key$ + "ssid", "")
DIM password$ = WORKCONF.VALUE(cfg, key$ + "password", "")
DIM security$ = WORKCONF.VALUE(cfg, key$ + "security", "WPA")
DIM hidden = WORKCONF.VALUE(cfg, key$ + "hidden", FALSE)
DIM n_cards = WORKCONF.VALUE(cfg, key$ + "cards", 8)
DIM home$ = "~/Documents/AutomateWork/print/guest_wifi.pdf"
DIM out$ = WORKCONF.FOLDER$(cfg, key$ + "out_file", home$)
IF ssid$ = "" THEN THROW "Set guest_wifi.ssid in work.conf first."

DIM payload$ = WIFICARD.PAYLOAD$(ssid$, password$, security$, hidden)
IF WORKCONF.FLAG("dry-run") THEN
    PRINT payload$
    PRINT QR.TEXT$(QR.MATRIX(payload$, {"ecc": "M"}))
ELSE
    MKDIR PATH.DIRNAME$(out$)
    DIM r = WIFICARD.SHEET(ssid$, password$, payload$, out$, n_cards)
    PRINT "Wrote "; r{"cards"}; " cards to"
    PRINT "    "; out$
ENDIF
```

The card module knows the Wi-Fi code format and draws the cards with the QR module on a PDFGEN page:

```

' wificard.jdb: cards with the guest Wi-Fi's name, password and a QR
' code that a phone camera reads to join the network.
'
'   IMPORT WIFICARD
'   DIM p$ = WIFICARD.PAYLOAD$("Guests", "sun;flower", "WPA", FALSE)
'   WIFICARD.SHEET("Guests", "sun;flower", p$, "wifi.pdf", 8)

EXPORT MODULE WIFICARD

IMPORT QR, PDFGEN

' A value as the Wi-Fi code format wants it: a backslash before each
' of \ ; , : and the double quote.
EXPORT FUNC ESCAPE$(s$)
    DIM out$ = ""
    DIM chars = -s$
    DIM k = 0
    FOR k = 0 TO LEN(chars) - 1
        DIM c$ = chars[k]
        IF INSTR("\;,:\" + CHR$(34), c$) >= 0 THEN out$ = out$ + "\"
        out$ = out$ + c$
    NEXT k
    RETURN out$
ENDFUNC

' The text of the QR code: security is "WPA" (also for WPA2 and WPA3),
' "WEP" or "nopass" for an open network; hidden adds H:true.
EXPORT FUNC PAYLOAD$(ssid$, password$, security$, hidden)
    DIM sec$ = UCASE$(TRIM$(security$))
    IF sec$ = "WPA2" ORELSE sec$ = "WPA3" THEN sec$ = "WPA"
    IF sec$ = "NOPASS" ORELSE sec$ = "" THEN sec$ = "nopass"
    IF NOT (sec$ IN ["WPA", "WEP", "nopass"]) THEN
        THROW "WIFICARD: no security called " + security$
    ENDIF
    DIM out$ = "WIFI:T:" + sec$ + ";S:" + ESCAPE$(ssid$) + ";"
    IF sec$ <> "nopass" THEN
        out$ = out$ + "P:" + ESCAPE$(password$) + ";"
    ENDIF
    IF hidden THEN out$ = out$ + "H:true;"
    RETURN out$ + ";"
ENDFUNC

' One card of 85 by 55 mm with its top left corner at x, y.
SUB Card(pdf, code, x, y, ssid$, password$)
    PDFGEN.SETDRAW(pdf, 190, 190, 190)
    PDFGEN.SETLINEWIDTH(pdf, 0.2)
    PDFGEN.BOX(pdf, x, y, 85, 55, "D")
    PDFGEN.SETCOLOR(pdf, 31, 56, 100)

```

```

PDFGEN.USEFONT(pdf, "helvetica", "B", 13)
PDFGEN.TEXTAT(pdf, x + 5, y + 10, "Guest Wi-Fi")
PDFGEN.SETCOLOR(pdf, 90, 90, 90)
PDFGEN.USEFONT(pdf, "helvetica", "", 7.5)
PDFGEN.TEXTAT(pdf, x + 5, y + 18, "Network")
PDFGEN.TEXTAT(pdf, x + 5, y + 30, "Password")
PDFGEN.SETCOLOR(pdf, 0, 0, 0)
PDFGEN.USEFONT(pdf, "helvetica", "B", 10)
PDFGEN.TEXTAT(pdf, x + 5, y + 23, PDFGEN.FIT$(pdf, ssid$, 40))
PDFGEN.USEFONT(pdf, "courier", "", 10)
DIM shown$ = password$
IF shown$ = "" THEN shown$ = "(none)"
PDFGEN.TEXTAT(pdf, x + 5, y + 35, PDFGEN.FIT$(pdf, shown$, 40))
PDFGEN.USEFONT(pdf, "helvetica", "I", 7)
PDFGEN.SETCOLOR(pdf, 90, 90, 90)
PDFGEN.TEXTAT(pdf, x + 5, y + 49, "Point your phone's camera")
PDFGEN.TEXTAT(pdf, x + 5, y + 52, "at the code to join.")
QR.TOPDF(pdf, code, x + 46, y + 8, 36, 2)
ENDSUB

' An A4 page with n_cards cards (1 to 8) to cut out. Answers "cards"
' and "modules", the QR code's size in modules per side.
EXPORT FUNC SHEET(ssid$, password$, payload$, path$, n_cards)
  IF n_cards < 1 ORELSE n_cards > 8 THEN
    THROW "WIFICARD: 1 to 8 cards fit on a page"
  ENDIF
  DIM code = QR.MATRIX(payload$, {"ecc": "M"})
  DIM size = QR.INFO(){"size"}
  DIM pdf = PDFGEN.DOC({"size": "A4", "margin": 15})
  PDFGEN.SETINFO(pdf, "Guest Wi-Fi " + ssid$, "", "Wi-Fi cards")
  PDFGEN.ADDPAGE(pdf)
  DIM k = 0
  FOR k = 0 TO n_cards - 1
    DIM x = 15 + (k MOD 2) * 95
    DIM y = 15 + (k \ 2) * 68
    Card(pdf, code, x, y, ssid$, password$)
  NEXT k
  PDFGEN.WRITEFILE(pdf, path$)
  RETURN {"cards": n_cards, "modules": size}
ENDFUNC

```

How it works

1. Phones read a Wi-Fi QR code as one line of text such as
WIFI:T:WPA;S:Miller Guests;P:Coffee\ :2026;;. T is the kind of security,
S the network's name, P the password.

2. In that line the characters \ ; , : and the double quote have a meaning of their own, so WIFICARD.ESCAPE\$ puts a backslash before each of them when they are part of the name or the password. Without it, the colon in Coffee:2026 would cut the password short.
3. WIFICARD.PAYLOAD\$ puts the parts together. WPA2 and WPA3 networks are written as WPA, which is what phones expect; an open network has no password at all.
4. WIFICARD.SHEET makes the QR code with QR.MATRIX once and draws it on every card with QR.TOPDF, next to the name and the password.

Run it

Try `--dry-run` first: it prints the code's text and the code itself in the terminal. Scan the screen with your phone to check it before you print:

```
jdbasic guest_wifi.jdb --dry-run
WIFI:T:WPA;S:Miller Guests;P:Coffee\:2026;;
```

Without the switch the program writes `guest_wifi.pdf` with eight cards:

```
jdbasic guest_wifi.jdb
Wrote 8 cards to
C:\Users\mia\Documents\AutomateWork\print\guest_wifi.pdf
```

Schedule it

Not at all: the cards stay right until the password changes. Run the program again after a change; the wizard leaves this recipe unscheduled.

Make it yours

The settings are in the `[guest_wifi]` part of `work.conf`:

```
[guest_wifi]
ssid = "Miller Guests"
password = "Coffee:2026"
security = "WPA"
cards = 8
out_file = "~/Documents/AutomateWork/print/guest_wifi.pdf"
```

- **One large card for the reception desk:** `cards = 1`.
- **An open network:** `security = "nopass"`; the card then shows (none) as the password.
- **A hidden network:** add `hidden = true`, and phones look for it even though it does not announce its name.

When it goes wrong

- **"Set guest_wifi.ssid":** the network's name is missing in `work.conf`.
- **The phone does not offer to join:** scan the `--dry-run` code and compare the text with the settings. A wrong security kind is the usual cause; WEP networks are rare today.
- **A character shows as ? on the card:** the card uses the PDF standard fonts, which know the letters of Western European languages. The QR code itself is still right.

Balance dividend

About 5 minutes a week of spelling passwords, and visitors who are on the network before the coffee arrives.

Bonus: E16 Form Filler

The chore

Twice a week Mia books a trip in the travel portal of her company. The form asks the same things every time: her name, her staff number, the cost centre, her manager. Only the destination and the dates change. She types the same five fields again and again, and once a month she mistypes the staff number and the booking comes back.

What you get

You click into the first field of a form and start the Form Filler. After a short countdown it types your values into the fields, one after the other, with Tab between them, the way you would. You check the form, add what is new this time, and send it yourself.

Before you start

The setup wizard has run. The Form Filler is a recipe you start by hand, so the wizard plans no time for it. You need the values of your form, in the order the form asks for them. To find that order, click into the first field and press Tab a few times: the cursor jumps from field to field the way the Form Filler will type.

Watch out

While it types, the Form Filler takes over your keyboard. Whatever window is in front gets the key strokes, so keep your hands off the keyboard and the mouse until it is done. To stop it at once, hold Esc, or push the mouse into the top-left corner of the screen. Never start it on a screen that others can see or use, and ask your IT department first: some companies forbid programs that type for you. Never put a password into the list. `work.conf` is a plain text file that anyone with access to your folder can read.

The program

The program reads your values and the countdown, builds the key strokes, and either shows them or types them:


```

' Form Filler: types the values you enter again and again into a form,
' with Tab between the fields. The values are the [form_filler] part
' of work.conf.
'
' jdbasic form_filler.jdb          fill the form after a countdown
' jdbasic form_filler.jdb --dry-run show what it would type

IMPORT WORKCONF, INPUTKIT, FORMFILL

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "form_filler."
DIM window$ = WORKCONF.VALUE(cfg, key$ + "window", "")
DIM values = WORKCONF.VALUE(cfg, key$ + "values", [])
DIM wait_s = WORKCONF.VALUE(cfg, key$ + "countdown", 5)
DIM submit = WORKCONF.VALUE(cfg, key$ + "submit", FALSE)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM job = FORMFILL.JOB(window$, values, submit)
DIM errors = job{"errors"}
IF LEN(errors) > 0 THEN
    PRINT "The settings do not make a form fill:"
    PRINT " "; JOIN(errors, CHR$(10) + " ")
    END 1
ENDIF
DIM steps = job{"steps"}
IF LEN(steps) = 0 THEN
    PRINT "Nothing to type. Add values to [form_filler] in work.conf."
    END
ENDIF
IF dry_run THEN
    DIM n = INPUTKIT.RUN_STEPS(steps, TRUE)
    PRINT n; " steps. Run without --dry-run to type them."
    END
ENDIF

IF window$ = "" THEN PRINT "Click into the first field of the form."
INPUTKIT.COUNTDOWN(wait_s)
TRY
    DIM done = INPUTKIT.RUN_STEPS(steps, FALSE)
    PRINT "Filled: "; done; " steps."
CATCH
    PRINT "Form Filler stopped: "; ERRMSG$
    END 1
ENDTRY

```

The module turns your values into a macro, a short list of commands such as type Mia Example and key tab:

```

' formfill.jdb: the key strokes that fill a form. The values go into
' the fields one after the other, with Tab between them, as a macro of
' INPUTKIT, so a dry run can show every step before anything is typed.
'
'   IMPORT FORMFILL
'   DIM job = FORMFILL.JOB("", ["Mia Example", "Sales"], FALSE)
'   PRINT JOIN(INPUTKIT.PLAN_LINES(job{"steps"}), CHR$(10))

EXPORT MODULE FORMFILL

IMPORT INPUTKIT

' The macro that types the values: a window line first when a title is
' given, a Tab after every value but the last, Enter at the end when
' submit is TRUE. An empty value leaves its field as it is.
EXPORT FUNC MACRO$(window$, values, submit)
    DIM lines = []
    DIM title$ = TRIM$(window$)
    IF title$ <> "" THEN PUSH(lines, "window " + title$)
    DIM k = 0
    FOR k = 0 TO LEN(values) - 1
        DIM v$ = values[k]
        IF v$ <> "" THEN PUSH(lines, "type " + v$)
        IF k < LEN(values) - 1 THEN PUSH(lines, "key tab")
    NEXT k
    IF submit AND ALSO LEN(values) > 0 THEN PUSH(lines, "key enter")
    RETURN JOIN(lines, CHR$(10))
ENDFUNC

' The fill as INPUTKIT steps: {"steps": [...], "errors": [...]}.
EXPORT FUNC JOB(window$, values, submit)
    RETURN INPUTKIT.PARSE(MACRO$(window$, values, submit))
ENDFUNC

```

The typing itself happens in the shared module `inputkit.jdb` in the folder `recipes/lib`, which the wizard copies with the other shared modules. Chapter 4 shows its macro language in the Macro Player (M16), and Chapter 5 its emergency stop in the Desktop Robot (X16).

How it works

1. `WORKCONF.VALUE` reads the `[form_filler]` part of `work.conf`: the values, the title of a window, the countdown and whether to press Enter at the end.
- 2.

FORMFILL.MACRO\$ writes one type line per value and a key tab line between two values. An empty value types nothing, so its Tab skips a field you want to leave as it is.

3. FORMFILL.JOB hands the macro to INPUTKIT.PARSE, which checks every line and turns it into a step.
4. With `--dry-run` the program prints the steps and ends. Without it, INPUTKIT.COUNTDOWN gives you a few seconds to click into the first field, and RUN_STEPS types the values.
5. Before every value and every key the module looks at Esc and at the position of the mouse. Either one stops the run with a message, and nothing more is typed.

Run it

With five values in `work.conf`, the third one empty, the dry run shows every key stroke before anything happens:

```
jdbasic form_filler.jdb --dry-run
would line 1: type 'Mia Example' (11 characters)
would line 2: press tab
would line 3: type '4711' (4 characters)
would line 4: press tab
would line 5: press tab
would line 6: type 'CC 3100 Sales' (13 characters)
would line 7: press tab
would line 8: type 'Anna Berg' (9 characters)
8 steps. Run without --dry-run to type them.
```

Open the form and start the program without `--dry-run`. The command prompt now has the focus, so click into the first field of the form during the countdown.

Schedule it

The Form Filler has no schedule: a form is there when you open it, not at a fixed time. Start it from a command prompt, from a shortcut on the desktop that runs `jdbasic form_filler.jdb`, or with a button of the Work Cockpit (X06).

Make it yours

The settings are the `[form_filler]` part of `work.conf`:

```
[form_filler]
values = ["Mia Example", "4711", "", "CC 3100 Sales", "Anna Berg"]
window = ""
countdown = 5
submit = false
```

- **More fields.** Add values to the list, in the order of the form. An empty value `""` leaves a field as it is.
- **The right window first.** Put the exact title of the form's window into `window`, as its title bar shows it, and the Form Filler brings that window to the front before it types. The cursor still has to be in the first field.
- **Send the form.** `submit = true` presses Enter after the last value. Leave it off until you have watched a few runs.
- **More time.** `countdown = 10` gives you ten seconds to click into the form.

When it goes wrong

- **A value lands in the wrong field:** the form's Tab order is not the order you see. Press Tab through the form by hand and count; add an empty value for each field you want to skip.
- **The text goes into another window:** the window that was in front when the countdown ended got it. Click into the form during the countdown, or name the form in `window`.
- **Nothing arrives, and there is no message:** the form belongs to a program that runs as administrator. Windows does not let a normal program type into it, and it does not say so. Start that program normally, or fill its form by hand.
- **"Esc is held: stopped.":** you pressed Esc, or a key got stuck. Run it again.

Balance dividend

About 15 minutes a week for two forms of five fields, and the staff number that is always right.

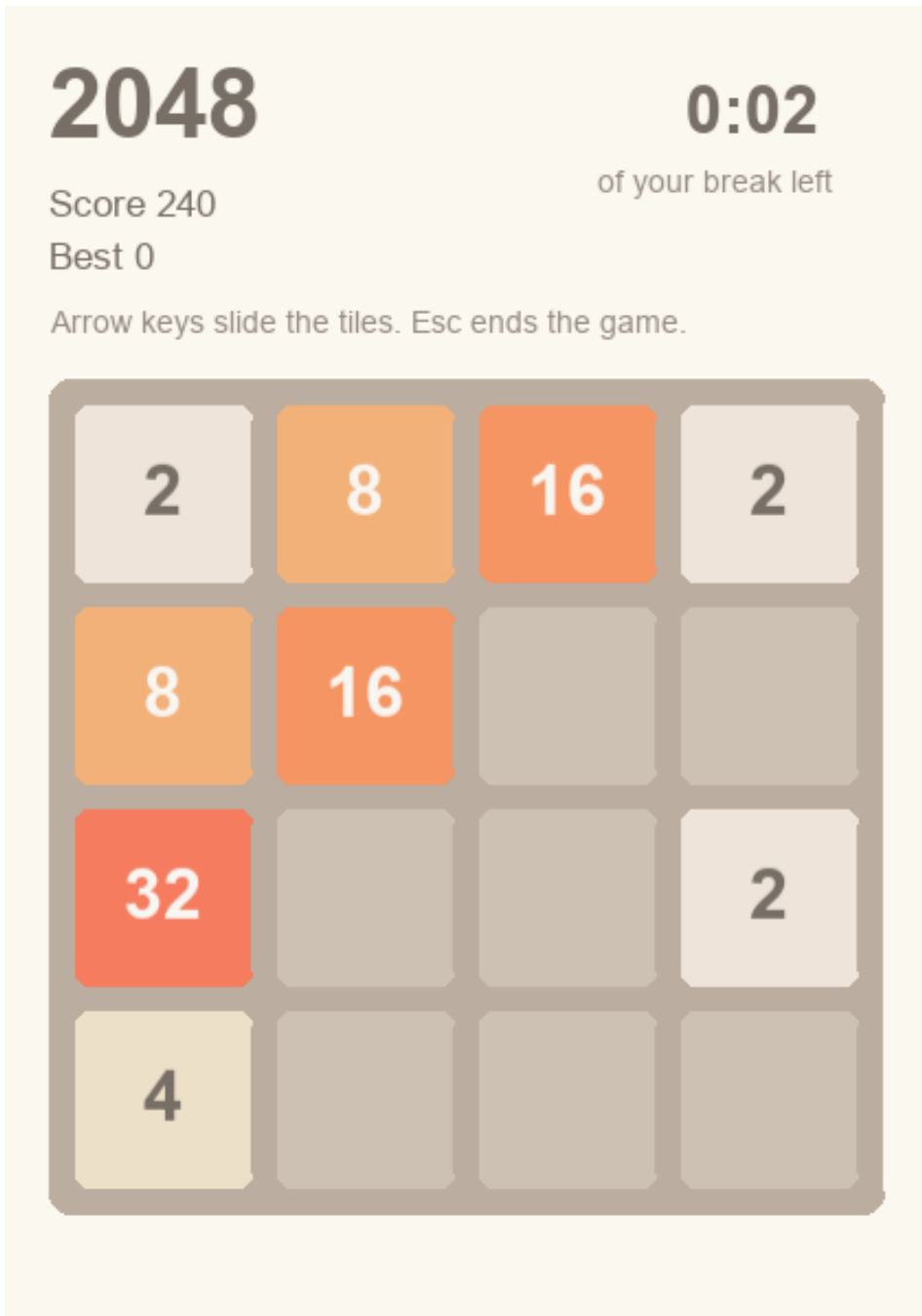
Bonus: E17 Break Arcade

The chore

The Break Reminder (E05) tells Mia when it is time for a break. She gets up, fetches a coffee, and is back at her desk two minutes later, where the inbox is already waiting. The break counts on paper, but her head never left work. What she needs is something that takes her mind off the job for a few minutes and then lets her go.

What you get

A round of 2048 in a window of its own. You slide numbered tiles with the arrow keys; two equal tiles join into one with twice the value, and every join scores. The game knows how long your break is. A clock counts it down, and when it reaches zero the game says the break is over, shows your score, keeps your best one, and closes.



A round of 2048 near the end of a break

Before you start

The setup wizard has run. The Break Arcade needs no settings to start: a break of three minutes and a file for your best score in the work folder are the defaults. It opens a graphics window, so it needs the release of jdBasic you installed in Chapter 2; nothing else.

The program

The program draws the board, reads the arrow keys, and watches the clock:

```
' Break Arcade: a round of 2048 for your break. The arrow keys slide
' the tiles; when the break is over, the game says so and closes. The
' settings are the [break_arcade] part of work.conf.
'
'   jdbasic break_arcade.jdb           play until the break ends
'   jdbasic break_arcade.jdb --selftest play itself and check
'   jdbasic break_arcade.jdb --selftest --shots out/arcade

IMPORT WORKCONF, CONF, ARCADE

DIM selftest = WORKCONF.FLAG("selftest")
DIM shots$ = WORKCONF.ARG$("shots")
DIM cfg = CONF.TOML_TEXT("")
IF NOT selftest THEN cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "break_arcade."
DIM minutes = WORKCONF.VALUE(cfg, key$ + "minutes", 3)
DIM home$ = "~/Documents/AutomateWork/arcade_best.txt"
DIM best_file$ = WORKCONF.FOLDER$(cfg, key$ + "best_file", home$)
IF selftest THEN best_file$ = MKTEMP$("arcade") + ".txt"

CONST W = 420
CONST H = 600
CONST GAP = 12
CONST CELL = 80
CONST TOP = 170
DIM bold$ = "C:/Windows/Fonts/arialbd.ttf"
DIM plain$ = "C:/Windows/Fonts/arial.ttf"

' The colour of a tile as [r, g, b].
FUNC TileColour(v)
    IF v = 0 THEN RETURN [205, 193, 180]
    IF v = 2 THEN RETURN [238, 228, 218]
    IF v = 4 THEN RETURN [237, 224, 200]
    IF v = 8 THEN RETURN [242, 177, 121]
    IF v = 16 THEN RETURN [245, 149, 99]
    IF v = 32 THEN RETURN [246, 124, 95]
```

```

    IF v = 64 THEN RETURN [246, 94, 59]
    IF v <= 512 THEN RETURN [237, 204, 97]
    RETURN [237, 194, 46]
ENDFUNC

SUB Font(bold, size)
    DIM path$ = plain$
    IF bold THEN path$ = bold$
    IF FILE.EXISTS(path$) THEN SETFONT path$, size
ENDSUB

' Text centred on x.
SUB Centred(x, y, t$, r, g, b)
    DIM size = GFX.TEXTSIZE(t$)
    TEXT x - size[0] / 2, y, t$, r, g, b
ENDSUB

SUB DrawGame(game, ms_left, best, note$)
    CLS 250, 248, 239
    Font(TRUE, 44)
    TEXT 20, 20, "2048", 119, 110, 101
    Font(FALSE, 17)
    TEXT 20, 80, "Score " + STR$(game{"score"}), 119, 110, 101
    TEXT 20, 104, "Best " + STR$(best), 119, 110, 101
    Font(TRUE, 30)
    TEXT W - 110, 30, ARCADE.CLOCK$(ms_left), 119, 110, 101
    Font(FALSE, 14)
    TEXT W - 150, 72, "of your break left", 150, 140, 130
    TEXT 20, 136, note$, 150, 140, 130
    ROUNDED_RECT 20, TOP, W - 40, W - 40, 10, TRUE, 187, 173, 160
    DIM board = game{"board"}
    DIM r = 0
    DIM c = 0
    FOR r = 0 TO 3
        DIM cells = board[r]
        FOR c = 0 TO 3
            DIM v = cells[c]
            DIM x = 20 + GAP + c * (CELL + GAP)
            DIM y = TOP + GAP + r * (CELL + GAP)
            DIM col = TileColour(v)
            ROUNDED_RECT x, y, CELL, CELL, 6, TRUE, _
                col[0], col[1], col[2]
            IF v > 0 THEN
                DIM size = 32
                IF v >= 128 THEN size = 26
                IF v >= 1024 THEN size = 21
                Font(TRUE, size)
                DIM tx = x + CELL / 2

```



```

        DIM ty = y + CELL / 2 - size * 0.6
        IF v <= 4 THEN
            Centred(tx, ty, STR$(v), 119, 110, 101)
        ELSE
            Centred(tx, ty, STR$(v), 249, 246, 242)
        ENDIF
    ENDIF
NEXT c
NEXT r
ENDSUB

SUB DrawEnd(game, best, new_best)
    CLS 250, 248, 239
    Font(TRUE, 34)
    Centred(W / 2, 170, "Break is over", 119, 110, 101)
    Font(FALSE, 20)
    DIM top$ = STR$(ARCADE.TOP_TILE(game{"board"}))
    DIM line$ = "Score " + STR$(game{"score"}) + ", top tile " + top$
    Centred(W / 2, 230, line$, 119, 110, 101)
    IF new_best THEN
        Centred(W / 2, 262, "A new best score!", 246, 124, 95)
    ELSE
        DIM so_far$ = "Best so far " + STR$(best)
        Centred(W / 2, 262, so_far$, 119, 110, 101)
    ENDIF
    Font(TRUE, 24)
    Centred(W / 2, 340, "Back to work, and well done.", 119, 110, 101)
ENDSUB

SUB Shot(name$)
    IF shots$ = "" THEN RETURN
    MKDIR shots$
    GFX.SAVE_SCREENSHOT(PATH.JOIN$(shots$, name$))
ENDSUB

DIM seed = (TICK() MOD 2147483645) + 1
IF selftest THEN seed = 7
DIM game = ARCADE.NEW_GAME(seed)
DIM best = ARCADE.BEST(best_file$)
DIM length_ms = minutes * 60000
IF selftest THEN length_ms = 4000
DIM script = ["left", "up", "right", "down"]
DIM fed = 0
DIM moves = 0
DIM over = FALSE

SCREEN W, H, "Break Arcade"
DIM started = TICK()

```

```

DO
    DIM left_ms = length_ms - (TICK() - started)
    IF left_ms <= 0 THEN EXITDO
    DIM k$ = INKEY$()
    IF k$ = CHR$(27) THEN EXITDO
    DIM dir$ = ARCADE.KEY_DIR$(k$)
    IF selftest AND ALSO fed < 40 THEN
        dir$ = script[feed MOD 4]
        fed = fed + 1
    ENDIF
    IF over AND ALSO k$ = CHR$(13) THEN
        game = ARCADE.NEW_GAME((TICK() MOD 2147483645) + 1)
        over = FALSE
    ENDIF
    IF dir$ <> "" AND ALSO NOT over THEN
        DIM after = ARCADE.TURN(game, dir$)
        IF after{"moved"} THEN moves = moves + 1
        game = after
        over = NOT ARCADE.CAN_MOVE(game{"board"})
    ENDIF
    DIM note$ = "Arrow keys slide the tiles. Esc ends the game."
    IF over THEN note$ = "No move left. Enter starts a new game."
    DrawGame(game, left_ms, best, note$)
    IF selftest AND ALSO fed = 38 THEN Shot("arcade_game.png")
    SCREENFLIP
    SLEEP 16
LOOP

DIM new_best = ARCADE.KEEP_BEST(best_file$, game{"score"})
DIM best_now = ARCADE.BEST(best_file$)
DrawEnd(game, best_now, new_best)
Shot("arcade_end.png")
SCREENFLIP
DIM pause_ms = 4000
IF selftest THEN pause_ms = 600
SLEEP pause_ms
GFX.CLOSE()
PRINT "Score "; game{"score"}; ", best "; best_now

IF selftest THEN
    DIM failed = 0
    IF moves < 10 THEN
        PRINT "FAIL: only "; moves; " moves changed the board"
        failed = failed + 1
    ENDIF
    IF game{"score"} <= 0 THEN
        PRINT "FAIL: no points after "; moves; " moves"
        failed = failed + 1
    ENDIF

```

```

ENDIF
IF best_now <> game{"score"} THEN
    PRINT "FAIL: the best score was not kept"
    failed = failed + 1
ENDIF
IF FILE.EXISTS(best_file$) THEN KILL best_file$
IF failed > 0 THEN END 1
PRINT "ALL ARCADE WINDOW TESTS PASSED"
ENDIF

```

The rules of the game are a module of their own. Each function takes a board and answers a new one, so a test can play any situation without a window:

```

' arcade.jdb: the rules of 2048 as plain functions. A board is a
' list of four rows of four numbers, 0 for an empty cell. A move
' slides every tile as far as it goes, joins two equal tiles into
' one, and scores the value of each new tile. Random tiles come from
' a seed, so a game can be played again exactly.
'
'   IMPORT ARCADE
'   DIM game = ARCADE.NEW_GAME(42)
'   DIM turn = ARCADE.MOVE(game{"board"}, "left")

EXPORT MODULE ARCADE

CONST SIZE = 4

' The next number of the random sequence, from 1 to 2147483646.
EXPORT FUNC NEXT_SEED(seed)
    DIM s = seed MOD 2147483647
    IF s <= 0 THEN s = s + 2147483646
    RETURN (s * 16807) MOD 2147483647
ENDFUNC

' An empty board.
EXPORT FUNC EMPTY_BOARD()
    DIM rows = []
    DIM r = 0
    FOR r = 1 TO SIZE
        PUSH(rows, [0, 0, 0, 0])
    NEXT r
    RETURN rows
ENDFUNC

' One row slid to the left: answers "row" and "gain", the points the
' joined tiles bring. A tile joins at most once per move.
EXPORT FUNC SLIDE_ROW(row)

```

```

DIM tiles = FILTER(LAMBDA x -> x <> 0, row)
DIM out = []
DIM gain = 0
DIM k = 0
DO WHILE k < LEN(tiles)
  DIM here = tiles[k]
  IF k + 1 < LEN(tiles) AND ALSO tiles[k + 1] = here THEN
    PUSH(out, here * 2)
    gain = gain + here * 2
    k = k + 2
  ELSE
    PUSH(out, here)
    k = k + 1
  ENDIF
LOOP
DO WHILE LEN(out) < LEN(row)
  PUSH(out, 0)
LOOP
RETURN {"row": out, "gain": gain}
ENDFUNC

' Whether two boards hold the same tiles.
EXPORT FUNC SAME(a, b)
  DIM r = 0
  DIM c = 0
  FOR r = 0 TO SIZE - 1
    DIM ra = a[r]
    DIM rb = b[r]
    FOR c = 0 TO SIZE - 1
      IF ra[c] <> rb[c] THEN RETURN FALSE
    NEXT c
  NEXT r
  RETURN TRUE
ENDFUNC

' One move of the whole board, dir$ "left", "right", "up" or "down".
' Answers "board", "gain" and "moved" (FALSE when nothing changed).
EXPORT FUNC MOVE(board, dir$)
  DIM d$ = LCASE$(dir$)
  IF NOT (d$ IN ["left", "right", "up", "down"]) THEN
    THROW "ARCADE: no direction " + dir$
  ENDIF
  DIM work = board
  IF d$ = "up" ORELSE d$ = "down" THEN work = TRANSPOSE(board)
  DIM rows = []
  DIM gain = 0
  DIM r = 0
  FOR r = 0 TO SIZE - 1

```

```

    DIM row = work[r]
    IF d$ = "right" ORELSE d$ = "down" THEN row = REVERSE(row)
    DIM slid = SLIDE_ROW(row)
    DIM new_row = slid{"row"}
    IF d$ = "right" ORELSE d$ = "down" THEN
        new_row = REVERSE(new_row)
    ENDIF
    PUSH(rows, new_row)
    gain = gain + slid{"gain"}
NEXT r
IF d$ = "up" ORELSE d$ = "down" THEN rows = TRANSPOSE(rows)
DIM result AS MAP
result{"board"} = rows
result{"gain"} = gain
result{"moved"} = NOT SAME(board, rows)
RETURN result
ENDFUNC

' The empty cells of a board as [row, column] pairs, row by row.
EXPORT FUNC EMPTY_CELLS(board)
    DIM out = []
    DIM r = 0
    DIM c = 0
    FOR r = 0 TO SIZE - 1
        DIM cells = board[r]
        FOR c = 0 TO SIZE - 1
            IF cells[c] = 0 THEN PUSH(out, [r, c])
        NEXT c
    NEXT r
    RETURN out
ENDFUNC

' A new tile in a random empty cell: a 2, or a 4 one time in ten.
' Answers "board" and "seed", the seed for the next tile.
EXPORT FUNC SPAWN(board, seed)
    DIM free = EMPTY_CELLS(board)
    DIM result AS MAP
    result{"board"} = board
    result{"seed"} = seed
    IF LEN(free) = 0 THEN RETURN result
    DIM s1 = NEXT_SEED(seed)
    DIM s2 = NEXT_SEED(s1)
    DIM spot = free[s1 MOD LEN(free)]
    DIM value = 2
    IF s2 MOD 10 = 0 THEN value = 4
    DIM rows = []
    DIM r = 0
    DIM c = 0

```

```

    FOR r = 0 TO SIZE - 1
        DIM old = board[r]
        DIM row = []
        FOR c = 0 TO SIZE - 1
            IF r = spot[0] AND ALSO c = spot[1] THEN
                PUSH(row, value)
            ELSE
                PUSH(row, old[c])
            ENDIF
        NEXT c
        PUSH(rows, row)
    NEXT r
    result{"board"} = rows
    result{"seed"} = s2
    RETURN result
ENDFUNC

' A new game: two tiles on an empty board. Answers "board", "seed"
' and "score".
EXPORT FUNC NEW_GAME(seed)
    DIM first = SPAWN(EMPTY_BOARD(), seed)
    DIM second = SPAWN(first{"board"}, first{"seed"})
    DIM game AS MAP
    game{"board"} = second{"board"}
    game{"seed"} = second{"seed"}
    game{"score"} = 0
    RETURN game
ENDFUNC

' One turn: the move, and a new tile when the move changed the board.
' Answers the game with "moved" added.
EXPORT FUNC TURN(game, dir$)
    DIM m = MOVE(game{"board"}, dir$)
    DIM after AS MAP
    after{"board"} = m{"board"}
    after{"seed"} = game{"seed"}
    after{"score"} = game{"score"} + m{"gain"}
    after{"moved"} = m{"moved"}
    IF m{"moved"} THEN
        DIM s = SPAWN(m{"board"}, game{"seed"})
        after{"board"} = s{"board"}
        after{"seed"} = s{"seed"}
    ENDIF
    RETURN after
ENDFUNC

' Whether any move can still change the board.
EXPORT FUNC CAN_MOVE(board)

```

```

    IF LEN(EMPTY_CELLS(board)) > 0 THEN RETURN TRUE
    DIM d = 0
    DIM dirs = ["left", "right", "up", "down"]
    FOR d = 0 TO 3
        DIM m = MOVE(board, dirs[d])
        IF m{"moved"} THEN RETURN TRUE
    NEXT d
    RETURN FALSE
ENDFUNC

' The largest tile on the board.
EXPORT FUNC TOP_TILE(board)
    RETURN MAX(SELECT(LAMBDA row -> MAX(row), board))
ENDFUNC

' The direction a key name stands for, "" for any other key.
EXPORT FUNC KEY_DIR$(key$)
    DIM k$ = UCASE$(key$)
    IF k$ = "LEFT" ORELSE k$ = "A" THEN RETURN "left"
    IF k$ = "RIGHT" ORELSE k$ = "D" THEN RETURN "right"
    IF k$ = "UP" ORELSE k$ = "W" THEN RETURN "up"
    IF k$ = "DOWN" ORELSE k$ = "S" THEN RETURN "down"
    RETURN ""
ENDFUNC

' The best score kept in a file, 0 when there is none yet.
EXPORT FUNC BEST(path$)
    IF NOT FILE.EXISTS(path$) THEN RETURN 0
    DIM t$ = TRIM$(TXTREADER$(path$))
    IF t$ = "" THEN RETURN 0
    RETURN VAL(t$)
ENDFUNC

' Keeps score as the best when it beats the one in the file; answers
' TRUE for a new best.
EXPORT FUNC KEEP_BEST(path$, score)
    IF score <= BEST(path$) THEN RETURN FALSE
    MKDIR PATH.DIRNAME$(path$)
    TXTWRITER path$, STR$(score) + CHR$(10)
    RETURN TRUE
ENDFUNC

' The text of the clock: minutes and seconds left, such as "2:05".
EXPORT FUNC CLOCK$(ms_left)
    DIM s = INT(ms_left / 1000)
    IF s < 0 THEN s = 0
    DIM sec = s MOD 60
    DIM sec$ = STR$(sec)

```

```

IF sec < 10 THEN sec$ = "0" + sec$
RETURN STR$(INT(s / 60)) + ":" + sec$
ENDFUNC

```

How it works

1. A board is a list of four rows of four numbers, 0 for an empty cell. ARCADE.SLIDE_ROW slides one row to the left: FILTER gathers the tiles that are not 0, the loop joins two equal neighbours into one, and zeros fill the rest. A tile that was just made by a join does not join again in the same move, which is what makes 2048 a game of planning.
2. ARCADE.MOVE uses that one function for all four directions. For a move to the right REVERSE turns each row around, it slides, and REVERSE turns it back. For up and down TRANSPOSE swaps rows and columns first.
3. After every move that changed something, ARCADE.SPAWN puts a 2, or now and then a 4, into a free cell. The cell comes from a seeded sequence of numbers, so the same seed always gives the same game. The test uses that to play a game twice and compare.
4. The main loop asks INKEY\$ for the last key, turns an arrow into a direction with ARCADE.KEY_DIR\$, and draws the board again. TICK gives the milliseconds since the game began; the difference to the length of the break is the clock at the top right.
5. When no move can change the board any more, ARCADE.CAN_MOVE says so, and Enter starts a new game. When the clock reaches zero, the game shows its last screen, ARCADE.KEEP_BEST writes the score into the file if it beats the old best, and the window closes.

Run it

```
jdbasic break_arcade.jdb
```

The window opens and the clock starts. Slide with the arrow keys or with W, A, S and D. Esc ends the game early; the best score still counts. With `--selftest` the game plays a round of its own, about four seconds long, checks that the moves scored and the best score was kept, and closes again. It is how the test of this recipe knows that the window works.

Schedule it

A game has no schedule; it is there when you want a break. Start it by hand, from a shortcut on the desktop that runs `jdbasic break_arcade.jdb`, or from a button of the Work Cockpit (X06). A good moment is right after the Break Reminder (E05) has asked you to stand up: stretch first, then play one round.

Make it yours

The settings are the `[break_arcade]` part of `work.conf`:

```
[break_arcade]
minutes = 3
best_file = "~/Documents/AutomateWork/arcade_best.txt"
```

- **A longer break.** `minutes = 10` gives you ten minutes, which is about one good game.
- **A fresh start.** Delete the file in `best_file` and the best score starts again from zero.
- **Other colours.** `TileColour` in the program holds the colour of every tile as red, green and blue from 0 to 255. Change a line and the next game uses it.
- **A bigger goal.** The game ends only when the clock does. If you would rather stop at a tile, compare `ARCADE.TOP_TILE` with 2048 in the main loop and end the round there.

When it goes wrong

- **The window opens and closes at once:** `jdBasic` found no graphics. Check that `jdbasic --version` lists *GFX*; the release of Chapter 2 has it.
- **The arrow keys do nothing:** another window has the focus. Click into the game window once.
- **The numbers look plain:** the game looks for the fonts Arial and Arial Bold of Windows and falls back to the built in font when they are missing. It still works the same.

Balance dividend

No minutes saved this time. The dividend is a break that really is one: three minutes in which work is not on your mind, and a clock that sends you back without guilt.

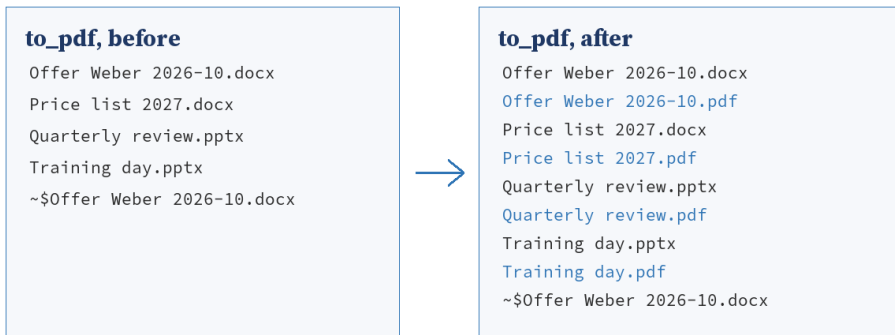
Bonus: E18 Office to PDF

The chore

Mia sends offers and price lists to customers, always as PDF, because a Word file looks different on every computer and invites changes. So every offer ends the same way: *File, Save As*, pick PDF, find the folder, click Save, and for the slides of a training day the same in PowerPoint. When she changes a price an hour later, she does it all again, and now and then the customer gets the PDF of the old version.

What you get

You keep your Word and PowerPoint files in one folder. Office to PDF puts a PDF next to each of them that has none yet, or whose PDF is older than the document. Word and PowerPoint make the PDFs themselves, so they look exactly like what you see on the screen. The programs work without a window and are closed again when the recipe is done.



The folder before and after a run: a PDF next to every document

Before you start

The setup wizard has run, and you have a folder for the documents, by default `to_pdf` in your work folder. Any folder works; many people point the recipe at the folder where their offers already live.

Watch out

The recipe needs Word and PowerPoint installed and licensed on this computer, the desktop versions of Microsoft Office. It starts them under your own account, the way you start them, so it runs from the Task Scheduler while you are logged in and not as a Windows service. A document that is open in Word while the recipe runs is still converted; the PDF shows the last saved state.

The program

The program reads the settings, asks the module which documents need a PDF and lets it make them:

```
' Office to PDF: makes a PDF of every Word and PowerPoint file in a
' folder that has none yet or has changed since, with the Office
' programs themselves, so the PDF looks exactly like the document.
' The settings are the [office_to_pdf] part of work.conf.
'
'   jdbasic office_to_pdf.jdb           make the PDFs
'   jdbasic office_to_pdf.jdb --dry-run  show which PDFs it would make

IMPORT WORKCONF, TOPDF

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "office_to_pdf."
DIM home$ = "~/Documents/AutomateWork/to_pdf"
DIM source$ = WORKCONF.FOLDER$(cfg, key$ + "source", home$)
DIM target$ = WORKCONF.FOLDER$(cfg, key$ + "target", "")
DIM wanted = ["word", "powerpoint"]
DIM programs = WORKCONF.VALUE(cfg, key$ + "programs", wanted)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM plan = TOPDF.PLAN(source$, target$, programs)
DIM n_new = TOPDF.TO_MAKE(plan)
IF n_new = 0 THEN
    PRINT "Every PDF in "; source$; " is up to date."
    END
ENDIF
DIM done = TOPDF.APPLY(plan, dry_run)
IF dry_run THEN
    PRINT n_new; " PDFs would be made. Run without --dry-run."
ELSE
    DIM n_skipped = LEN(done{"failed"})
```

```
PRINT done{"made"}; " PDFs made, "; n_skipped; " skipped."
ENDIF
```

The module plans the work first and then talks to Office:

```
' topdf.jdb: the work of Office to PDF. PLAN looks at a folder and
' decides which documents need a new PDF; APPLY makes them with Word,
' PowerPoint or Excel, each started once and without a window.
'
'   IMPORT TOPDF
'   DIM plan = TOPDF.PLAN("C:/Users/mia/Offers", "", ["word"])
'   DIM done = TOPDF.APPLY(plan, FALSE)

EXPORT MODULE TOPDF

IMPORT OFFICEKIT

' The documents of a folder as maps with "src", "pdf", "kind" and
' "action": "convert" when the PDF is missing or older, "current"
' when it is up to date. Only the kinds in programs count; Office's
' own lock files (~$name) are left out. out_dir$ "" means the folder
' itself.
EXPORT FUNC PLAN(folder$, out_dir$, programs)
  DIM out = []
  IF NOT FILE.ISDIR(folder$) THEN RETURN out
  DIM target$ = out_dir$
  IF target$ = "" THEN target$ = folder$
  DIM names = SORT(DIR$(PATH.JOIN$(folder$, "*")))
  DIM k = 0
  FOR k = 0 TO LEN(names) - 1
    DIM file_name$ = names[k]
    DIM src$ = PATH.JOIN$(folder$, file_name$)
    DIM kind$ = OFFICEKIT.KIND$(file_name$)
    DIM wanted = Listed(programs, kind$)
    DIM is_lock = STARTSWITH(file_name$, "~$")
    DIM is_dir = FILE.ISDIR(src$)
    IF wanted AND ALSO NOT is_lock AND ALSO NOT is_dir THEN
      DIM pdf$ = OFFICEKIT.PDF_PATH$(src$, target$)
      DIM item AS MAP
      item{"src"} = src$
      item{"pdf"} = pdf$
      item{"kind"} = kind$
      item{"action"} = "current"
      IF OFFICEKIT.NEEDS_PDF(src$, pdf$) THEN
        item{"action"} = "convert"
      ENDIF
      PUSH(out, item)
```

```

        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' Whether a kind is one of the programs.
FUNC Listed(programs, kind$)
    IF kind$ = "" THEN RETURN FALSE
    RETURN kind$ IN SELECT(LAMBDA p$ -> LCASE$(p$), programs)
ENDFUNC

' How many documents of a plan need a PDF.
EXPORT FUNC TO_MAKE(plan)
    RETURN LEN(FILTER(LAMBDA i -> i{"action"} = "convert", plan))
ENDFUNC

' Makes the PDFs of a plan, or with dry_run only prints them. Every
' Office program is started once and quit at the end. Answers
' {"made": n, "failed": [lines]}; a document Office cannot open is
' named and left out.
EXPORT FUNC APPLY(plan, dry_run)
    DIM made = 0
    DIM failed = []
    DIM kinds = ["word", "powerpoint", "excel"]
    DIM j = 0
    DIM k = 0
    FOR j = 0 TO LEN(kinds) - 1
        DIM kind$ = kinds[j]
        DIM todo = FILTER(LAMBDA USE(kind$) i -> _
            i{"kind"} = kind$ AND ALSO i{"action"} = "convert", plan)
        IF LEN(todo) > 0 THEN
            IF dry_run THEN
                FOR k = 0 TO LEN(todo) - 1
                    DIM one = todo[k]
                    PRINT "would make "; PATH.BASENAME$(one{"pdf"})
                NEXT k
            ELSE
                DIM r = Convert(kind$, todo)
                made = made + r{"made"}
                DIM lost = r{"failed"}
                FOR k = 0 TO LEN(lost) - 1
                    DIM why$ = lost[k]
                    PUSH(failed, why$)
                NEXT k
            ENDIF
        ENDIF
    NEXT j
    RETURN {"made": made, "failed": failed}

```

```

ENDFUNC

' The PDFs of one Office program, with the program started once.
FUNC Convert(kind$, todo)
    DIM made = 0
    DIM failed = []
    DIM prog$ = OFFICEKIT.PROG_ID$(kind$)
    DIM app = CREATEOBJECT(prog$)
    OFFICEKIT.PREPARE(app, prog$)
    DIM k = 0
    FOR k = 0 TO LEN(todo) - 1
        DIM item = todo[k]
        DIM src$ = item{"src"}
        DIM pdf$ = item{"pdf"}
        TRY
            MKDIR PATH.DIRNAME$(pdf$)
            OFFICEKIT.SAVE_PDF(app, kind$, src$, pdf$)
            PRINT "made "; PATH.BASENAME$(pdf$)
            made = made + 1
        CATCH
            PRINT "skipped "; ERRMSG$
            PUSH(failed, ERRMSG$)
        ENDTRY
    NEXT k
    OFFICEKIT.CLOSE_APP(app, prog$)
    RETURN {"made": made, "failed": failed}
ENDFUNC

```

How it works

1. TOPDF.PLAN lists the folder and keeps the files Word (.docx, .doc, .rtf, .odt) and PowerPoint (.pptx, .ppt, .odp) open. Files starting with ~\$ are Office's own lock files and are left out.
2. For each document, OFFICEKIT.NEEDS_PDF compares the time it was last saved with the time of its PDF. A missing or older PDF means convert, a newer one current.
3. TOPDF.APPLY groups the documents by program, so Word starts once for all Word files and PowerPoint once for all slides. For each program FILTER keeps the documents of that kind that need a PDF; USE(kind\$) hands the lambda the program it looks for. CREATEOBJECT starts the program, and OFFICEKIT.PREPARE hides its window and turns off its questions.
- 4.

OFFICEKIT.SAVE_PDF opens the document read only, so it is never changed, saves it as PDF and closes it again. A document that Office cannot open is named in a skipped line and the others go on.

5. OFFICEKIT.CLOSE_APP quits the program, and RELEASEOBJECT lets go of it, so no Word stays behind in the background.

The library OFFICEKIT in recipes/lib holds what the Office recipes share; Excel Refresh (M18) and the Outlook Bridge (X19) use it too.

Run it

See what would happen first:

```
jdbasic office_to_pdf.jdb --dry-run
would make Offer Weber 2026-10.pdf
would make Price list 2027.pdf
would make Quarterly review.pdf
would make Training day.pdf
4 PDFs would be made. Run without --dry-run.
```

Then without --dry-run. You see a made line per PDF, and a few seconds per document. A second run right after it says that every PDF is up to date.

Schedule it

The wizard plans it daily at noon. If you change documents all day long, an hourly run works as well, since a run with nothing to do starts no Office program at all.

Make it yours

The settings are the [office_to_pdf] part of work.conf:

```
[office_to_pdf]
source = "~/Documents/AutomateWork/to_pdf"
target = ""
programs = ["word", "powerpoint"]
```

- **All PDFs in one place.** Set target to a folder, for example the one you share with your colleagues, and the PDFs land there instead of next to the documents.

- **Workbooks too.** Add "excel" to programs, and every workbook becomes a PDF of all its sheets. Excel Refresh (M18) does more for a single report workbook.
- **Only Word.** Leave out "powerpoint" when your slides are not meant for customers.

When it goes wrong

- **"Every PDF is up to date" although you changed a document:** the document was not saved yet. Save it and run again.
- **"skipped Offer.docx: ...":** Word could not open the file, most often because it is damaged or protected by a password. Open it by hand to see what Word says.
- **"COM: Cannot find ProgID 'Word.Application'"** at the start: Word or PowerPoint is not installed on this computer, or only the web version is. The recipe needs the desktop programs.
- **A WINWORD.EXE stays in the Task Manager:** the recipe was stopped in the middle, for example with Ctrl+C. End that process in the Task Manager; the next run starts a fresh one.

Balance dividend

About 15 minutes a week for anyone who sends documents as PDF, and no more customers who get the PDF of yesterday's version.

Chapter 4: Medium: Fifteen Programs for Documents, Mail and Data

Jonas leads a team of eight. His week is made of sheets: the hours of his people, the budgets of three projects, the invoices for two customers, the vacation plan, the Friday status that the head of the department reads on Monday. None of it is hard. All of it takes time, and most of it is copying numbers from one place to another and checking that nothing went missing on the way.

The recipes of this chapter take over that kind of work. They read and write the files an office runs on: Excel workbooks, Word documents, PDF, calendars, mail. Each one does a job that is worth an hour or two a week, and each one is a program of 80 to 200 lines that you will read and change.

What changes at the medium level

At the easy level you changed settings. Here you change code. The recipes are written so that the parts you are likely to change sit together and are easy to find, and every recipe's *Make it yours* section names the lines and shows the change. You do not need to understand every line before you start; you need to know where the line is that decides the thing you want different.

Three habits make that safe:

1. **Copy before you change.** Keep the recipe folder as it came, work on a copy, and switch the wizard over when the copy works. If you know git, a repository for your work folder is better still.

2. **Run the test.** Every recipe comes with a file that ends in `_test.jdb`. Run it after each change with `jdbasic` like any other program. It prints one line per check and ends with how many passed. A change that breaks a check shows you the line and what it expected.
3. **Run it dry.** As before, `--dry-run` shows what the program would do. With real documents of your own, look at that list before you let it act.

Mail without passwords in files

Several recipes of this chapter write mail: invoices, letters, the Friday status, the timesheet for your boss. None of them stores a password, in `work.conf` or anywhere else. They follow one rule instead:

- A recipe that runs on its own builds the messages and puts them as `.eml` files into the folder `outbox` in your work folder. Any mail program opens such a file, so you can look at each message before it goes out.
- When you run the recipe yourself with `--send`, it asks for your mail password, sends what is in the outbox, and moves every sent message into `outbox/sent`. The prompt shows a star for each key, so nobody reads the password over your shoulder.

The password lives in memory for the few seconds the sending takes and is never written down. If you would rather not type it every time, you can keep it in the Windows Credential Manager, where Windows stores it encrypted for your account: add a generic credential named `AutomateWork/mail` with your mail user name and password. The recipes look there first and only ask when they find nothing. Recipe X14 in Chapter 5 shows how to do the same from the command prompt. A password with letters such as `ä` or `é` is safer there too, because the console prompt reads plain keys only.

If your company does not allow programs to send mail with your account at all, the outbox alone still saves most of the work: open each file and press *Send* in your mail program.

The mail server settings go into `work.conf` once, in the part that belongs to you:

```
[mail]
server = "smtps://mail.example.com:465"
user = "jonas@example.com"
```

```
from = "Jonas Example <jonas@example.com>"
```

Your IT department knows the server name and port; many offices use port 465 with `smtps`, or 587 with `smtp` and `STARTTLS`.

The recipes

Recipe	What it takes off your desk
M01 Invoice Generator	Invoices from the rows of a sheet, as Word or PDF, ready to mail
M02 Report Merger	The monthly sheets of the team in one workbook with totals
M03 Mail Merge	Personal letters from a Word template and a contact list
M04 Receipt Sorter	Receipts filed by month and category, and the expense report
M05 Shift and Vacation Planner	The team calendar as a file every calendar program reads
M06 Friday Status Mail	The weekly status from your notes, as a mail to the team
M07 Inbox Unpacker	Attachments of saved mails filed by sender and subject
M08 Personal Dashboard	Your numbers on one page in the browser
M09 Log Detective	Errors per hour from a log file, as a chart
M10 Hot Folder	Whatever lands in a folder processed by rules
M11 Data Cleaner	Messy exports made consistent, with a report of what changed
M12 Meeting Cost Meter	What your meetings cost in hours and money
M13 Snippet Tool	The text blocks you type ten times a day
M14 Contract Diff	A readable report of what changed between two Word files
M15 Timesheet for the Boss	The monthly timesheet from the time tracker, mailed
M16 Macro Player (bonus)	Clicks and keystrokes for programs that offer nothing else
M17 Slide Builder (bonus)	A PowerPoint presentation from an outline or a sheet
M18 Excel Refresh (bonus)	The Monday report refreshed in Excel itself, formulas and charts intact

The order is a suggestion, not a path. Each recipe stands on its own; where one builds on another, such as the timesheet on the time tracker of E07, the text says so.

M01 Invoice Generator

The chore

Jonas's team sells supplies to a handful of regular customers. At the end of every month someone writes the invoices: open last month's Word file, change the number, type the customer, copy the lines from the order sheet, add up, add the VAT, save as PDF, write a mail, attach, send. Eight customers, eight times the same twenty steps, and once a quarter a number is used twice or a total does not match the lines.

What you get

You keep the month's orders in one Excel sheet, one row per item. The Invoice Generator turns the rows of each customer into one invoice with the next free number, the lines, the net amount, the VAT and the total, as a PDF or a Word file. For every customer with a mail address it puts a ready mail with the invoice attached into the outbox.

Example Supplies

1 Main Street, Exametown

Miller Ltd

Invoice RE-2026-0001

Date: 2026-10-01

Item	Quantity	Price	Amount
Copy paper	10	4.50 EUR	45.00 EUR
Delivery	1	12.00 EUR	12.00 EUR
Net			57.00 EUR
VAT 19 %			10.83 EUR
Total			67.83 EUR

The top of an invoice the generator writes

Before you start

The setup wizard has run, and the outbox rule of this chapter is set up, with your name and address as from in [mail]. You need one Excel file with a sheet like this one; the order of the columns does not matter, their names do:

Customer	Email	Item	Quantity	Price
Miller Ltd	buy@miller.example	Copy paper	10	4.50
Stone and Co	office@stone.example	Toner	2	61.90
Miller Ltd		Delivery	1	12.00

A customer can have as many rows as there are items. The mail address is taken from the customer's first row.

The program

The program reads its settings, the sheet and the counter, and then either shows the invoices, writes them, or sends the mails that wait:

```
' Invoice Generator: turns the rows of an Excel sheet into numbered
' invoices as PDF or Word files, and puts a mail for each one into the
' outbox. The settings are the [invoice_generator] part of work.conf,
' the mail server and the outbox the [mail] part.
'
'   jdbasic invoice_generator.jdb           write invoices and mails
'   jdbasic invoice_generator.jdb --dry-run show the invoices only
'   jdbasic invoice_generator.jdb --send    send what waits in outbox

IMPORT WORKCONF, XLSX, INVOICES, INVOICE_PDF, INVOICE_WORD, OUTBOX

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "invoice_generator."
DIM work$ = "~/Documents/AutomateWork/"
DIM sheet$ = WORKCONF.FOLDER$(cfg, key$ + "sheet", _
    work$ + "invoices.xlsx")
DIM tab$ = WORKCONF.VALUE(cfg, key$ + "tab", "")
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", _
    work$ + "invoices")
DIM prefix$ = WORKCONF.VALUE(cfg, key$ + "prefix", "RE-")
DIM kind$ = LCASE$(WORKCONF.VALUE(cfg, key$ + "format", "pdf"))
DIM vat = WORKCONF.VALUE(cfg, key$ + "vat_percent", 19)
DIM sender$ = OUTBOX.FROM$(cfg)
DIM company AS MAP
```

```

company{"name"} = WORKCONF.VALUE(cfg, key$ + "company", "")
company{"address"} = WORKCONF.VALUE(cfg, key$ + "address", "")
company{"date"} = FORMAT_DATE(NOW(), "%Y-%m-%d")

IF WORKCONF.FLAG("send") THEN
    DIM waiting = OUTBOX.PENDING(cfg)
    PRINT LEN(waiting); " mails wait in "; OUTBOX.FOLDER$(cfg)
    IF LEN(waiting) > 0 THEN
        DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
        PRINT result{"sent"}; " sent."
        DIM p = 0
        FOR p = 0 TO LEN(result{"failed"}) - 1
            PRINT " not sent: "; result{"failed"}[p]
        NEXT p
    ENDIF
END
ENDIF

DIM book = XLSX.READ(sheet$)
IF tab$ = "" THEN tab$ = XLSX.SHEETS(sheet$)[0]
IF NOT MAP.EXISTS(book, tab$) THEN THROW "No sheet " + tab$
DIM list = INVOICES.GROUP(book{tab$})
DIM year$ = FORMAT_DATE(NOW(), "%Y")
DIM counter$ = PATH.JOIN$(folder$, "last_number.txt")
DIM numbers = INVOICES.NEXT_NUMBERS(counter$, prefix$, year$, _
    LEN(list))
DIM dry_run = WORKCONF.FLAG("dry-run")
IF NOT dry_run THEN MKDIR folder$

DIM k = 0
FOR k = 0 TO LEN(list) - 1
    DIM inv = INVOICES.TOTALS(list[k], vat)
    DIM number$ = numbers[k]
    DIM total$ = INVOICES.MONEY$(inv{"gross"})
    DIM shown$ = number$ + " " + inv{"customer"} + " " + total$
    IF dry_run THEN
        PRINT "would write "; shown$
    ELSE
        DIM file$ = PATH.JOIN$(folder$, number$ + "." + kind$)
        IF kind$ = "docx" THEN
            INVOICE_WORD.WRITE(inv, number$, company, file$)
        ELSE
            INVOICE_PDF.WRITE(inv, number$, company, file$)
        ENDIF
    ENDIF
    IF inv{"email"} <> "" THEN
        DIM m = INVOICES.MAIL_FOR(inv, number$, sender$, file$)
        OUTBOX.PUT$(m, cfg, number$ + ".eml")
    ENDIF

```



```

        PRINT "wrote "; shown$
    ENDIF
NEXT k

IF NOT dry_run AND ALSO LEN(list) > 0 THEN
    DIM last$ = numbers[LEN(numbers) - 1]
    TXTWRITER counter$, STR$(VAL(RIGHT$(last$, 4)))
    PRINT "The mails wait in the outbox. Send them with --send."
ENDIF
IF LEN(list) = 0 THEN PRINT "No rows in "; tab$

```

The module INVOICES does the work that does not depend on where the data comes from: grouping, adding up, numbering, the rows of the table and the mail:

```

' invoices.jdb: invoices from the rows of a sheet. Rows with the same
' customer become one invoice with one line per row; each invoice gets
' the next number, its totals, the rows of its table and a mail. The
' PDF and the Word file are written by INVOICE_PDF and INVOICE_WORD.
'
'   IMPORT INVOICES
'   DIM list = INVOICES.GROUP(rows)
'   DIM inv = INVOICES.TOTALS(list[0], 19)

EXPORT MODULE INVOICES

IMPORT MAIL

' The position of each wanted column in the header row, by its name
' in any case; a missing column is an error that names it.
EXPORT FUNC COLUMNS(header)
    DIM wanted = ["customer", "email", "item", "quantity", "price"]
    DIM out AS MAP
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(wanted) - 1
        out{wanted[k]} = -1
        FOR j = 0 TO LEN(header) - 1
            DIM h$ = LCASE$(TRIM$(" " + header[j]))
            IF h$ = wanted[k] THEN out{wanted[k]} = j
        NEXT j
        IF out{wanted[k]} < 0 THEN _
            THROW "The sheet has no column " + wanted[k]
    NEXT k
    RETURN out
ENDFUNC

```

```

' One invoice per customer, in the order they first appear: maps with
' "customer", "email" and "lines" (maps with item, quantity, price).
EXPORT FUNC GROUP(rows)
  DIM col = COLUMNS(rows[0])
  DIM out = []
  DIM where AS MAP
  DIM k = 0
  FOR k = 1 TO LEN(rows) - 1
    DIM r = rows[k]
    DIM who$ = TRIM$("" + r[col{"customer"}])
    IF who$ <> "" THEN
      IF NOT MAP.EXISTS(where, who$) THEN
        where{who$} = LEN(out)
        DIM mail$ = TRIM$("" + r[col{"email"}])
        PUSH(out, {"customer": who$, "email": mail$, _
          "lines": []})
      ENDIF
      DIM inv = out[where{who$}]
      DIM item$ = "" + r[col{"item"}]
      DIM qty = VAL("" + r[col{"quantity"}])
      DIM price = VAL("" + r[col{"price"}])
      DIM inv_lines = inv{"lines"}
      PUSH(inv_lines, {"item": item$, "quantity": qty, _
        "price": price})
    ENDIF
  NEXT k
  RETURN out
ENDFUNC

' The invoice with "net", "vat" and "gross" added, each rounded to
' cents; vat_percent such as 19.
EXPORT FUNC TOTALS(inv, vat_percent)
  DIM net = SUM(SELECT(LAMBDA ln -> ln{"quantity"} * ln{"price"}, _
    inv{"lines"}))
  inv{"net"} = ROUND(net * 100) / 100
  inv{"vat"} = ROUND(net * vat_percent) / 100
  inv{"gross"} = inv{"net"} + inv{"vat"}
  inv{"vat_percent"} = vat_percent
  RETURN inv
ENDFUNC

' The next n invoice numbers after the one stored in the counter file,
' such as RE-2026-0007; the file is not changed.
EXPORT FUNC NEXT_NUMBERS(counter$, prefix$, year$, n)
  DIM last = 0
  IF FILE.EXISTS(counter$) THEN _
    last = VAL(TRIM$(TXTREADER$(counter$)))

```

```

    RETURN SELECT(LAMBDA USE(prefix$, year$) v -> _
        prefix$ + year$ + "-" + RIGHT$("000" + STR$(v), 4), _
        last + IOTA(n))
ENDFUNC

EXPORT FUNC MONEY$(v)
    RETURN FORMAT$(".2f", v) + " EUR"
ENDFUNC

' The rows of the invoice table, the header first.
EXPORT FUNC TABLE_ROWS(inv)
    DIM body = SELECT(LAMBDA ln -> _
        [ln{"item"}, STR$(ln{"quantity"}), MONEY$(ln{"price"}), _
        MONEY$(ln{"quantity"} * ln{"price"})], inv{"lines"})
    RETURN APPEND([["Item", "Quantity", "Price", "Amount"]], body)
ENDFUNC

EXPORT FUNC TOTAL_LINES(inv)
    DIM vat$ = "VAT " + STR$(inv{"vat_percent"}) + " %"
    RETURN [["Net", MONEY$(inv{"net"})], [vat$, MONEY$(inv{"vat"})], _
        ["Total", MONEY$(inv{"gross"})]]
ENDFUNC

' The mail that goes with an invoice, the file attached.
EXPORT FUNC MAIL_FOR(inv, number$, sender$, attachment$)
    DIM subject$ = "Invoice " + number$
    DIM m = MAIL.MESSAGE(sender$, inv{"email"}, subject$)
    DIM body$ = "Hello," + CHR$(10) + CHR$(10)
    body$ = body$ + "please find invoice " + number$ + " attached, "
    body$ = body$ + "total " + MONEY$(inv{"gross"}) + "." + CHR$(10)
    body$ = body$ + CHR$(10) + "Kind regards" + CHR$(10)
    MAIL.PLAIN(m, body$ + MAIL.ADDRESSNAME$(sender$))
    MAIL.ATTACH(m, attachment$)
    RETURN m
ENDFUNC

```

Two small modules turn an invoice into a file. `INVOICE_PDF` places it on a page; `INVOICE_WORD` does the same with `DOCX` and is in the recipe folder:

```

' invoice_pdf.jdb: an invoice of the INVOICES module as an A4 PDF
' page.
'
'   IMPORT INVOICE_PDF
'   INVOICE_PDF.WRITE(inv, "RE-2026-0007", company, "x.pdf")

EXPORT MODULE INVOICE_PDF

```

```

IMPORT PDFGEN, INVOICES

' The invoice as a PDF page: company, customer, number and date, the
' table and the totals. company holds "name", "address" and "date".
EXPORT SUB WRITE(inv, number$, company, path$)
    DIM pdf = PDFGEN.DOC()
    PDFGEN.SETINFO(pdf, "Invoice " + number$, company{"name"})
    PDFGEN.ADDPAGE(pdf)
    PDFGEN.USEFONT(pdf, "helvetica", "B", 14)
    PDFGEN.TEXTAT(pdf, 20, 25, company{"name"})
    PDFGEN.USEFONT(pdf, "helvetica", "", 9)
    PDFGEN.TEXTAT(pdf, 20, 31, company{"address"})
    PDFGEN.USEFONT(pdf, "helvetica", "", 11)
    PDFGEN.TEXTAT(pdf, 20, 52, inv{"customer"})
    PDFGEN.USEFONT(pdf, "helvetica", "B", 16)
    PDFGEN.TEXTAT(pdf, 20, 72, "Invoice " + number$)
    PDFGEN.USEFONT(pdf, "helvetica", "", 10)
    PDFGEN.TEXTAT(pdf, 20, 79, "Date: " + company{"date"})
    PDFGEN.SETXY(pdf, 20, 88)
    DIM rows = INVOICES.TABLE_ROWS(inv)
    DIM opts = {"headers": rows[0], "widths": [0, 22, 32, 32], _
        "align": ["L", "R", "R", "R"]}
    PDFGEN.TABLE(pdf, DROP(1, rows), opts)
    PDFGEN.LN(pdf, 4)
    DIM totals = INVOICES.TOTAL_LINES(inv)
    DIM k = 0
    FOR k = 0 TO LEN(totals) - 1
        IF k = 2 THEN PDFGEN.USEFONT(pdf, "helvetica", "B", 10)
        PDFGEN.SETXY(pdf, 118, PDFGEN.POSY(pdf))
        PDFGEN.CELL(pdf, 40, 6, totals[k][0])
        PDFGEN.CELL(pdf, 32, 6, totals[k][1], "R")
        PDFGEN.LN(pdf, 6)
    NEXT k
    PDFGEN.WRITEFILE(pdf, path$)
ENDSUB

```

How it works

1. **Settings.** The program reads its part of work.conf. Every value has a default, so a short [invoice_generator] part works. The sender comes from the shared mail settings through OUTBOX.FROM\$.
2. **The sheet.** XLSX.READ reads the whole workbook into a map from sheet name to rows. Without a tab setting the program takes the first sheet. Numbers in the sheet arrive as numbers, text as text.

3. **Finding the columns.** COLUMNS looks at the first row and notes where each of the five column names stands. That is why the order of the columns does not matter, and why a renamed column stops the program with a message that names it instead of producing wrong invoices.
4. **Grouping.** GROUP walks the rows and keeps a map from customer name to the position of that customer's invoice. The first row of a customer starts an invoice; every further row adds a line to it. Rows without a customer, such as an empty line at the end, are left out.
5. **Adding up.** TOTALS multiplies quantity and price of every line in one SELECT, adds the results with SUM, and rounds to cents. The VAT is computed from the unrounded net amount and rounded once, which is how a calculator would do it.
6. **Numbers.** NEXT_NUMBERS reads the last number used from `last_number.txt` in the invoice folder and hands out the next ones: `last + IOTA(n)` is the list of the next `n` numbers, and a SELECT turns each into a text such as RE-2026-0007. The program writes the new last number back only after all invoices are written, so a run that stops halfway does not burn numbers.
7. **Files.** INVOICE_PDF.WRITE places the parts on an A4 page with PDFGEN: your company at the top, the customer, the number and the date, the table of lines and the three totals. INVOICE_WORD.WRITE writes the same content as a Word file, for customers who want to edit it. Both take their rows from TABLE_ROWS, which makes one row per line with a SELECT, and from TOTAL_LINES, so the two files always agree.
8. **Mail.** MAIL_FOR builds the message with the invoice attached, and OUTBOX.PUT\$ writes it into the outbox under the invoice's number.

Run it

Look first:

```
jdbasic invoice_generator.jdb --dry-run
would write RE-2026-0001  Miller Ltd  67.83  EUR
would write RE-2026-0002  Stone and Co 147.32 EUR
```

Then write:

```
jdbasic invoice_generator.jdb
wrote RE-2026-0001 Miller Ltd 67.83 EUR
wrote RE-2026-0002 Stone and Co 147.32 EUR
The mails wait in the outbox. Send them with --send.
```

Open one of the .eml files in the outbox to check it, then send all of them:

```
jdbasic invoice_generator.jdb --send
```

Schedule it

Invoices are something you start yourself when the month's sheet is complete, so the wizard plans no task for this recipe. If your sheet is always complete on the first of the month, give it a schedule such as `daily 07:00` and let it run only on the first (see *Make it yours*).

Make it yours

The settings:

```
[invoice_generator]
sheet = "~/Documents/AutomateWork/invoices.xlsx"
tab = "October"
folder = "~/Documents/AutomateWork/invoices"
prefix = "RE-"
format = "pdf"
vat_percent = 19
company = "Example Supplies"
address = "1 Main Street, Exampletown"
```

`format = "docx"` writes Word files instead of PDF. One sheet per month, with `tab` set to the month, keeps last month's rows from being invoiced twice.

Changes in the code:

- **A line under the totals**, such as your bank details: in `INVOICE_PDF.WRITE`, after the loop that writes the totals, add `PDFGEN.TEXTAT(pdf, 20, PDFGEN.POSY(pdf) + 10, "IBAN ...")`.
- **Run only on the first of the month**: at the top of the program, after the settings, add `IF FORMAT_DATE(NOW(), "%d") <> "01" THEN END`, and schedule it daily.

- **Numbers that start again every year:** in NEXT_NUMBERS, keep one counter file per year by naming it "last_" + year\$ + ".txt" in the program, where counter\$ is set.
- **No VAT for some customers:** add a column VAT to the sheet, read it in GROUP like the other columns, store it in the invoice map, and pass inv{"vat_rate"} instead of vat to TOTALS.
- **The mail text:** it is built in MAIL_FOR, four short lines you can change freely.

When it goes wrong

- **"The sheet has no column price":** a column was renamed or the first row is not the header. The five names must appear in the first row, in any case.
- **Numbers jumped:** the counter in last_number.txt is the only memory of the generator. Do not delete it; to start over, write the number you want to come before the next one into it.
- **A customer got two invoices:** the name was written two ways, such as Miller Ltd and Miller Ltd.. Grouping compares names exactly.
- **The mail is missing:** the customer's first row has no address. The invoice file is written all the same.

Balance dividend

About 40 minutes a month for eight customers, ten a week, and no invoice number used twice.

M02 Report Merger

The chore

Every month the eight people of Jonas's team send a small Excel report: the projects they worked on, the hours, the costs, a note. Jonas opens eight files, copies the rows into one workbook, sorts them by project, adds up the hours with a formula that he fixes each time because the row count changed, and then notices that one report is missing and another one has a column more than the rest.

What you get

You put the reports of a month into one folder. The Report Merger reads all of them and writes one workbook: a sheet *Total* with the rows of all reports added up by project, and one sheet per person with the report as it came. Before it writes, it tells you who has not sent a report and whose columns differ from the others.

Before you start

The setup wizard has run. Make a folder for the reports with one subfolder per month, named like 2026-10. Each person's file goes in there under the person's name, such as `Anna.xlsx`. All reports have the same columns, and the first column names the row:

Project	Hours	Costs	Note
Alpha	10	250.50	on track
Beta	4	80.00	

The first column can be anything that the rows of different people have in common: a project, a cost centre, a customer.

The program

The program finds the month, reads the folder, reports what is missing, and writes the workbook:


```

' Report Merger: collects the monthly Excel reports of a team from one
' folder into a single workbook with a sheet of totals, and names who
' has not sent a report yet. The settings are the [report_merger] part
' of work.conf.
'
'   jdbasic report_merger.jdb                               last month
'   jdbasic report_merger.jdb --month 2026-10               a given month
'   jdbasic report_merger.jdb --dry-run                     show, write nothing

IMPORT WORKCONF, MERGER

' The month before the one of the given date, as YYYY-MM.
FUNC LastMonth$(day)
    DIM y = VAL(FORMAT_DATE(day, "%Y"))
    DIM m = VAL(FORMAT_DATE(day, "%m")) - 1
    IF m = 0 THEN
        m = 12
        y = y - 1
    ENDIF
    RETURN STR$(y) + "-" + RIGHT$("0" + STR$(m), 2)
ENDFUNC

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "report_merger."
DIM root$ = WORKCONF.FOLDER$(cfg, key$ + "folder", _
    "~/Documents/AutomateWork/reports")
DIM tab$ = WORKCONF.VALUE(cfg, key$ + "tab", "")
DIM team = WORKCONF.VALUE(cfg, key$ + "team", [])
DIM month$ = WORKCONF.ARG$("month")
IF month$ = "" THEN month$ = LastMonth$(NOW())

DIM folder$ = PATH.JOIN$(root$, month$)
IF NOT FILE.ISDIR(folder$) THEN THROW "No folder " + folder$
DIM reports = MERGER.REPORTS(folder$, tab$)
PRINT LEN(reports); " reports for "; month$

DIM missing = MERGER.MISSING(reports, team)
IF LEN(missing) > 0 THEN PRINT "Missing: "; JOIN(missing, ", ")
DIM odd = MERGER.ODD_HEADERS(reports)
IF LEN(odd) > 0 THEN
    PRINT "Other columns than the first report: "; JOIN(odd, ", ")
ENDIF
IF LEN(reports) = 0 THEN END

DIM totals = MERGER.TOTALS(reports)
DIM out$ = PATH.JOIN$(root$, month$ + " team.xlsx")
IF WORKCONF.FLAG("dry-run") THEN
    PRINT "would write "; out$; " with "; LEN(totals) - 1; " rows"

```

```

ELSE
    DIM sheets = MERGER.WRITE(out$, reports, totals)
    PRINT "wrote "; out$; " ("; sheets; " sheets)"
ENDIF

```

The module MERGER reads, compares, adds up and writes:

```

' merger.jdb: the monthly sheets of a team in one workbook. Every
' Excel file in a folder is one person's report; the rows are added up
' by their first column into a sheet of totals, and each report is
' kept on a sheet of its own.
'
'   IMPORT MERGER
'   DIM reports = MERGER.REPORTS("reports/2026-10", "")
'   DIM totals = MERGER.TOTALS(reports)
'   MERGER.WRITE("team.xlsx", reports, totals)

EXPORT MODULE MERGER

IMPORT XLSX

' The reports in a folder, sorted by file name: maps with "person"
' (the file name without .xlsx) and "rows" (the sheet tab$, or the
' first sheet when tab$ is "").
EXPORT FUNC REPORTS(folder$, tab$)
    DIM out = []
    DIM names = SORT(DIR$(PATH.JOIN$(folder$, "*.xlsx")))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM file$ = names[k]
        IF LEFT$(file$, 2) <> "~$" THEN
            DIM path$ = PATH.JOIN$(folder$, file$)
            DIM book = XLSX.READ(path$)
            DIM sheet$ = tab$
            IF sheet$ = "" THEN sheet$ = XLSX.SHEETS(path$)[0]
            IF NOT MAP.EXISTS(book, sheet$) THEN
                THROW file$ + " has no sheet " + sheet$
            ENDIF
            DIM who$ = LEFT$(file$, LEN(file$) - 5)
            PUSH(out, {"person": who$, "rows": book{sheet$}})
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' The names of the team that have no report in the folder.
EXPORT FUNC MISSING(reports, team)

```

```

    DIM have = SELECT(LAMBDA r -> LCASE$(r{"person"}), reports)
    RETURN FILTER(LAMBDA USE(have) w$ -> NOT (LCASE$(w$) IN have), _
        team)
ENDFUNC

' The reports whose header row differs from the first one's.
EXPORT FUNC ODD_HEADERS(reports)
    IF LEN(reports) = 0 THEN RETURN []
    DIM first$ = LCASE$(JOIN(reports[0]{"rows"}[0], "|"))
    DIM odd = FILTER(LAMBDA USE(first$) r -> _
        LCASE$(JOIN(r{"rows"}[0], "|")) <> first$, DROP(1, reports))
    RETURN SELECT(LAMBDA r -> r{"person"}, odd)
ENDFUNC

' The totals: the header of the first report, then one row per value
' of the first column in the order it first appears. Number cells are
' added up, any other cell keeps the first value seen.
EXPORT FUNC TOTALS(reports)
    IF LEN(reports) = 0 THEN RETURN []
    DIM header = reports[0]{"rows"}[0]
    DIM n_cols = LEN(header)
    DIM out = [header]
    DIM where AS MAP
    DIM k = 0
    DIM r = 0
    DIM c = 0
    FOR k = 0 TO LEN(reports) - 1
        DIM rows = reports[k]{"rows"}
        FOR r = 1 TO LEN(rows) - 1
            DIM row = rows[r]
            DIM key$ = TRIM$(" " + row[0])
            IF key$ <> "" THEN
                IF NOT MAP.EXISTS(where, key$) THEN
                    where{key$} = LEN(out)
                    DIM fresh = [key$]
                    FOR c = 1 TO n_cols - 1
                        PUSH(fresh, "")
                    NEXT c
                    PUSH(out, fresh)
                ENDIF
                DIM target = out[where{key$}]
                FOR c = 1 TO n_cols - 1
                    IF c < LEN(row) THEN
                        DIM cell = row[c]
                        DIM have = target[c]
                        IF TYPEOF(cell) <> "STRING" THEN
                            IF TYPEOF(have) = "STRING" THEN have = 0
                            target[c] = have + cell
                        ENDIF
                    ENDIF
                NEXT c
            ENDIF
        NEXT r
    NEXT k

```

```

        ELSEIF TYPEOF(have) = "STRING" THEN
            IF have = "" THEN target[c] = cell
        ENDIF
    ENDIF
NEXT c
ENDIF
NEXT r
NEXT k
RETURN out
ENDFUNC

' A sheet name Excel accepts: at most 31 characters, none of
' : \ / ? * [ ].
EXPORT FUNC SHEET_NAME$(name$)
    DIM t$ = REGEX.REPLACE("[:\\/?*\\[\\]", name$, "_")
    RETURN LEFT$(t$, 31)
ENDFUNC

' Writes the workbook: "Total" first, then one sheet per report.
' Answers the number of sheets.
EXPORT FUNC WRITE(path$, reports, totals)
    DIM wb = XLSX.NEW()
    DIM s = XLSX.SHEET(wb, "Total", totals)
    XLSX.COLUMN(s, "A", {"width": 24})
    XLSX.FREEZE(s, 1)
    DIM k = 0
    FOR k = 0 TO LEN(reports) - 1
        DIM rep = reports[k]
        DIM tab$ = SHEET_NAME$(rep{"person"})
        DIM ps = XLSX.SHEET(wb, tab$, rep{"rows"})
        XLSX.COLUMN(ps, "A", {"width": 24})
    NEXT k
    XLSX.WRITE(path$, wb)
    RETURN LEN(reports) + 1
ENDFUNC

```

How it works

1. **The month.** Without `--month` the program takes last month, which is what you want on the first working days of a month. `LastMonth$` handles January by going back to December of the year before.
2. **Reading.** `REPORTS` lists the `.xlsx` files of the month's folder in name order and reads each one with `XLSX.READ`. The person is the file name without `.xlsx`. Files that start with `~$` are the lock files Excel leaves while a workbook is open; they are skipped.

3. **Who is missing.** MISSING turns the reports into a list of their names in small letters with SELECT, and keeps with FILTER the people of the team list from the settings who are not in it. An empty list means that nobody is checked.
4. **Odd columns.** ODD_HEADERS joins each header row into one text and keeps with FILTER the reports whose text differs from the first report's; a SELECT answers their names. A report with a column more, a column less or the columns in another order is named, and its numbers still count by position.
5. **Adding up.** TOTALS keeps a map from the first column to the position of its row in the totals. The first time a project appears it gets a row of empty cells; after that every number in a report row is added to the cell in the same column. A text cell, such as the note, keeps the first text that is not empty.
6. **Writing.** WRITE puts the totals on the sheet *Total*, freezes its header row, and adds one sheet per report. SHEET_NAME\$ replaces the characters Excel does not allow in sheet names and cuts the name to 31 characters.

Run it

```
jdbasic report_merger.jdb --month 2026-10 --dry-run
3 reports for 2026-10
Missing: Dora
would write reports\2026-10 team.xlsx with 3 rows
```

Dora's report is missing. Once it is in the folder, run it again without `--dry-run`:

```
jdbasic report_merger.jdb --month 2026-10
4 reports for 2026-10
wrote reports\2026-10 team.xlsx (5 sheets)
```

The workbook lands next to the month's folder, so the folder itself keeps only the reports.

Schedule it

The merge is worth running when the reports are in, and that is a date nobody can plan. The wizard sets up no task for this recipe. Run it with `--dry-run` on the first working day to see who is missing, and without it when they have all sent theirs.

Make it yours

The settings:

```
[report_merger]
folder = "~/Documents/AutomateWork/reports"
tab = ""
team = ["Anna", "Ben", "Cleo", "Dora"]
```

tab names the sheet to read in each report. Leave it empty to read the first sheet, whatever its name.

Changes in the code:

- **Add up by two columns**, such as project and task: in TOTALS, build the key from both cells, DIM key\$ = TRIM\$(" " + row[0]) + " / " + TRIM\$(" " + row[1]). Start a new row as [row[0], row[1]] instead of [key\$], and let both FOR c = 1 loops start at 2.
- **Leave out a sum row** that people add at the bottom of their report: in TOTALS, change IF key\$ <> "" THEN to IF key\$ <> "" AND ALSO LCASE\$(key\$) <> "sum" THEN.
- **Stop on odd columns** instead of only naming them: in the program, after the line that prints them, add THROW "Fix the columns first." inside the IF.
- **The workbook in another folder**, such as a shared drive: change out\$ in the program to PATH.JOIN\$("S:/Team", month\$ + ".xlsx").
- **Mail the workbook** to the head of the department: build a message with MAIL.MESSAGE and MAIL.ATTACH(m, out\$) and put it into the outbox with OUTBOX.PUT\$, following the outbox rule of this chapter; M01 shows the lines.

When it goes wrong

- **"No folder ...\2026-09"**: the month's folder does not exist yet, or it is named another way. Give the month with --month, or name the folder YYYY-MM.
- **"Anna.xlsx has no sheet October"**: tab names a sheet that this report does not have. Leave tab empty, or ask for the template's sheet name to be kept.
-

A person is missing although the file is there: the file name and the name in team differ, such as `anna_mueller.xlsx` and Anna. Use the same name in both places.

- **A project appears twice in the totals:** it was written two ways, such as Alpha and Alpha . Spaces at the ends are removed; any other difference makes a new row.
- **Writing fails:** the workbook of the last run is still open in Excel, which locks the file. Close it and run again.

Balance dividend

About an hour a month of copying and checking, fifteen minutes a week, and the missing report is named before anyone asks.

M03 Mail Merge

The chore

Twice a year Jonas's team invites its customers: to the customer day in November, to a training in spring. The letter is always the same except for the name, the company and the salutation. Word has a mail merge, but it hides in a menu that nobody remembers between two uses, and it does not write one file per customer that can go out as an attachment. So someone copies the letter forty times and changes three lines in each.

What you get

You write the letter once in Word, with placeholders such as `{{name}}` where the personal parts go, and keep the contacts in an Excel sheet. The Mail Merge writes one Word file per contact with the placeholders filled, named after the person. For every contact with a mail address it puts a mail with the letter attached into the outbox.

Before you start

The setup wizard has run, and the outbox rule of this chapter is set up. You need a contact list with a header row; each column name is a placeholder:

Name	Salutation	Company	City	Email	Date
Anna Berg	Ms Berg	Stone and Co	Leeds	anna@stone.example	14 November
Ben Cole	Mr Cole	Cole and Sons	York		14 November

No template yet? Run the program once. It writes a sample letter to the template path and stops, so you can change that one in Word.

The program

The program reads its settings, the template and the contacts, and writes a letter and a mail per contact:


```

' Mail Merge: fills a Word template with the rows of a contact list,
' one letter per contact, and puts a mail with the letter into the
' outbox for every contact with an address. The settings are the
' [mail_merge] part of work.conf, the mail server the [mail] part.
,
'   jdbasic mail_merge.jdb           write letters and mails
'   jdbasic mail_merge.jdb --dry-run  show who gets a letter
'   jdbasic mail_merge.jdb --send    send what waits in the outbox

IMPORT WORKCONF, XLSX, MAIL, LETTERS, OUTBOX

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "mail_merge."
DIM work$ = "~/Documents/AutomateWork/"
DIM template$ = WORKCONF.FOLDER$(cfg, key$ + "template", _
    work$ + "templates/letter.docx")
DIM list$ = WORKCONF.FOLDER$(cfg, key$ + "contacts", _
    work$ + "contacts.xlsx")
DIM tab$ = WORKCONF.VALUE(cfg, key$ + "tab", "")
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", _
    work$ + "letters")
DIM pattern$ = WORKCONF.VALUE(cfg, key$ + "file_name", "{{name}}")
DIM send_mail = WORKCONF.VALUE(cfg, key$ + "mail", TRUE)
DIM subject$ = WORKCONF.VALUE(cfg, key$ + "subject", _
    "Invitation to our customer day")
DIM body$ = WORKCONF.VALUE(cfg, key$ + "body", _
    "Dear {{salutation}}," + CHR$(10) + CHR$(10) + _
    "please find our letter attached." + CHR$(10))
body$ = body$ + CHR$(10) + "Kind regards" + CHR$(10)
DIM sender$ = OUTBOX.FROM$(cfg)
body$ = body$ + MAIL.ADDRESSNAME$(sender$)

IF WORKCONF.FLAG("send") THEN
    DIM waiting = OUTBOX.PENDING(cfg)
    PRINT LEN(waiting); " mails wait in "; OUTBOX.FOLDER$(cfg)
    IF LEN(waiting) > 0 THEN
        DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
        PRINT result{"sent"}; " sent."
        DIM p = 0
        FOR p = 0 TO LEN(result{"failed"}) - 1
            PRINT "  not sent: "; result{"failed"}[p]
        NEXT p
    ENDIF
END
ENDIF

IF NOT FILE.EXISTS(template$) THEN
    DIM home$ = PATH.DIRNAME$(template$)

```

```

    IF home$ <> "" THEN MKDIR home$
    LETTERS.SAMPLE_TEMPLATE(template$)
    PRINT "No template yet. A sample is in "; template$
    PRINT "Change it in Word and run again."
    END
ENDIF

DIM book = XLSX.READ(list$)
IF tab$ = "" THEN tab$ = XLSX.SHEETS(list$)[0]
IF NOT MAP.EXISTS(book, tab$) THEN THROW "No sheet " + tab$
DIM people = LETTERS.CONTACTS(book{tab$})
DIM unfilled = LETTERS.UNFILLED(template$, people)
IF LEN(unfilled) > 0 THEN
    PRINT "No column for: "; JOIN(unfilled, ", ")
ENDIF

DIM dry_run = WORKCONF.FLAG("dry-run")
IF NOT dry_run THEN MKDIR folder$
DIM used AS MAP
DIM mails = 0
DIM k = 0
FOR k = 0 TO LEN(people) - 1
    DIM person = people[k]
    DIM file$ = LETTERS.FILE_NAME$(pattern$, person)
    IF MAP.EXISTS(used, LCASE$(file$)) THEN
        file$ = LEFT$(file$, LEN(file$) - 5) + " " + _
            STR$(k + 1) + ".docx"
    ENDIF
    used{LCASE$(file$)} = 1
    DIM address$ = ""
    IF MAP.EXISTS(person, "email") THEN address$ = person{"email"}
    DIM mailed = send_mail ANDALSO address$ <> ""
    DIM shown$ = file$
    IF mailed THEN shown$ = shown$ + " to " + address$
    IF dry_run THEN
        PRINT "would write "; shown$
    ELSE
        DIM path$ = PATH.JOIN$(folder$, file$)
        LETTERS.WRITE(template$, path$, person)
        IF mailed THEN
            DIM m = LETTERS.MAIL_FOR(person, sender$, subject$, _
                body$, path$)
            OUTBOX.PUT$(m, cfg, LEFT$(file$, LEN(file$) - 5) + ".eml")
            mails = mails + 1
        ENDIF
        PRINT "wrote "; shown$
    ENDIF
NEXT k

```

```

IF LEN(people) = 0 THEN PRINT "No contacts in "; tab$
IF mails > 0 THEN
    PRINT mails; " mails wait in the outbox. Send them with --send."
ENDIF

```

The module LETTERS turns rows into contacts and contacts into letters and mails:

```

' letters.jdb: personal letters from a Word template and a contact
' list. Each {{name}} in the template is filled from the column of the
' same name; each contact gets one Word file and, with an address, a
' mail with the letter attached.
'
'   IMPORT LETTERS
'   DIM people = LETTERS.CONTACTS(rows)
'   LETTERS.WRITE("letter.docx", "out/Anna.docx", people[0])

EXPORT MODULE LETTERS

IMPORT DOCX, MAIL

' The contacts of a sheet: one map per row from the column names in
' lower case to the cell texts. Rows without any text are left out.
EXPORT FUNC CONTACTS(rows)
    DIM out = []
    IF LEN(rows) = 0 THEN RETURN out
    DIM keys = SELECT(LAMBDA h -> LCASE$(TRIM$(" " + h)), rows[0])
    DIM k = 0
    DIM c = 0
    FOR k = 1 TO LEN(rows) - 1
        DIM person AS MAP
        DIM filled = FALSE
        FOR c = 0 TO LEN(keys) - 1
            DIM cell$ = ""
            IF c < LEN(rows[k]) THEN _
                cell$ = TRIM$(CellText$(rows[k][c]))
            person{keys[c]} = cell$
            IF cell$ <> "" THEN filled = TRUE
        NEXT c
        IF filled THEN PUSH(out, person)
    NEXT k
    RETURN out
ENDFUNC

' A cell as text; whole numbers without a decimal point.
FUNC CellText$(v)

```

```

    IF TYPEOF(v) = "STRING" THEN RETURN v
    IF v = INT(v) THEN RETURN STR$(INT(v))
    RETURN STR$(v)
ENDFUNC

' The placeholders of the template that no column of the contacts
' fills.
EXPORT FUNC UNFILLED(template$, contacts)
    IF LEN(contacts) = 0 THEN RETURN []
    DIM first = contacts[0]
    RETURN FILTER(LAMBDA USE(first) nm$ -> _
        NOT MAP.EXISTS(first, LCASE$(nm$)), _
        DOCX.PLACEHOLDERS(template$))
ENDFUNC

' A text with its {{name}} placeholders filled from the contact.
EXPORT FUNC FILL$(text$, person)
    RETURN REDUCE(LAMBDA USE(person) acc$, key$ -> _
        REPLACE$(acc$, "{" + key$ + "}", person{key$}), _
        MAP.KEYS(person), text$)
ENDFUNC

' A file name from the pattern, without characters Windows forbids.
EXPORT FUNC FILE_NAME$(pattern$, person)
    DIM t$ = TRIM$(FILL$(pattern$, person))
    t$ = REGEX.REPLACE("[<>:\x22/\\|?*]", t$, "_")
    IF t$ = "" THEN t$ = "letter"
    RETURN t$ + ".docx"
ENDFUNC

' Writes the letter of one contact, matching placeholders in any
' case; answers the number of placeholders filled.
EXPORT FUNC WRITE(template$, path$, person)
    DIM values AS MAP
    DIM wanted = DOCX.PLACEHOLDERS(template$)
    DIM k = 0
    FOR k = 0 TO LEN(wanted) - 1
        DIM key$ = LCASE$(wanted[k])
        IF MAP.EXISTS(person, key$) THEN
            values{wanted[k]} = person{key$}
        ENDIF
    NEXT k
    RETURN DOCX.FILL(template$, path$, values)
ENDFUNC

' The mail that goes with a letter; subject and text are filled from
' the contact like the template.
EXPORT FUNC MAIL_FOR(person, sender$, subject$, body$, attachment$)

```

```

    DIM m = MAIL.MESSAGE(sender$, person{"email"}, _
        FILL$(subject$, person))
    MAIL.PLAIN(m, FILL$(body$, person))
    MAIL.ATTACH(m, attachment$)
    RETURN m
ENDFUNC

' A template to start from, with the placeholders the sample contact
' list fills.
EXPORT SUB SAMPLE_TEMPLATE(path$)
    DIM doc = DOCX.NEW()
    DOCX.LANGUAGE(doc, "en-GB")
    DOCX.PARAGRAPH(doc, "{{name}}")
    DOCX.PARAGRAPH(doc, "{{company}}")
    DOCX.PARAGRAPH(doc, "{{city}}")
    DOCX.PARAGRAPH(doc, "")
    DOCX.PARAGRAPH(doc, "Dear {{salutation}},")
    DOCX.PARAGRAPH(doc, "thank you for working with us this " + _
        "year. We would like to invite you to our customer day " + _
        "on {{date}}.")
    DOCX.PARAGRAPH(doc, "Kind regards")
    DOCX.WRITE(doc, path$)
ENDSUB

```

How it works

1. **Contacts.** CONTACTS reads the header row once and turns every further row into a map from column name to cell text. The names are made lower case, so Email, EMAIL and email are the same column. A number in the sheet, such as a postcode, becomes text without a decimal point.
2. **Checking the template.** UNFILLED asks DOCX.PLACEHOLDERS for the placeholders in the template and keeps with FILTER the ones no column fills. The letters are written all the same; the placeholder stays in them as it is, which you will notice when you open one.
3. **Filling.** WRITE builds the values for DOCX.FILL from the contact. DOCX.FILL replaces each {{name}} in the body, the headers and the footers, also where Word has split it into several runs after an edit, and keeps all formatting of the template.
4. **File names.** FILE_NAME\$ fills the file_name pattern with FILL\$, which goes through the columns of the contact with REDUCE and replaces one placeholder after the other, and then replaces the characters Windows does not

allow in file names. Two contacts with the same name get the row number added, so no letter overwrites another.

5. **Mails.** MAIL_FOR fills the subject and the text from the same contact, attaches the letter, and the program puts the message into the outbox under the name of the letter.

Run it

Look first:

```
jdbasic mail_merge.jdb --dry-run
would write Anna Berg.docx to anna@stone.example
would write Ben Cole.docx
would write Cleo Dunn.docx to cleo@miller.example
```

Ben has no address; he gets a letter for the post. Then write:

```
jdbasic mail_merge.jdb
wrote Anna Berg.docx to anna@stone.example
wrote Ben Cole.docx
wrote Cleo Dunn.docx to cleo@miller.example
2 mails wait in the outbox. Send them with --send.
```

Open two or three of the letters before you send:

```
jdbasic mail_merge.jdb --send
```

Schedule it

A mail merge belongs to an occasion, not to a day of the week, so the wizard plans no task for it.

Make it yours

The settings:

```
[mail_merge]
template = "~/Documents/AutomateWork/templates/letter.docx"
contacts = "~/Documents/AutomateWork/contacts.xlsx"
tab = ""
```

```

folder = "~/Documents/AutomateWork/letters"
file_name = "{{company}} {{name}}"
subject = "Invitation to our customer day"
body = "Dear {{salutation}},\n\nplease see the letter attached.\n"
mail = true

```

file_name, subject and body take placeholders like the template. With mail = false the program writes only the letters, for post.

Changes in the code:

- **Only some contacts**, such as those with yes in a column Invite: in the program, after the line that reads people, keep only the contacts with yes:

```

people = FILTER(LAMBDA p -> LCASE$(p{"invite"}) = "yes", people)

```

- **A placeholder that is not a column**, such as today's date: in the program, before the loop, add DIM today\$ = FORMAT_DATE(NOW(), "%d %B %Y"), and at the top of the loop person{"today"} = today\$. The template can then use {{today}}.
- **A different letter per group**: add a column Template with a file name, and in the loop pass PATH.JOIN\$(PATH.DIRNAME\$(template\$), person{"template"}) to LETTERS.WRITE instead of template\$.
- **Copy to the account manager**: in MAIL_FOR, after MAIL.MESSAGE, add MAIL.CC(m, person{"manager"}) and a column Manager to the sheet.

When it goes wrong

- **"No column for: date"**: the template has a placeholder that the sheet does not fill. Add the column, or remove the placeholder.
- **A placeholder was not replaced** although the column exists: in the template it is written with a space or another bracket, such as {{ name }} or {name}. Write it exactly as {{name}}.
- **"No sheet ..."**: tab names a sheet the contact list does not have. Leave tab empty to use the first sheet.
- **The letters look odd in Word**: the formatting comes only from the template. Change fonts and spacing there; the program copies it as it is.

Balance dividend

Two invitations a year to forty customers, about two hours each time by hand, and a few smaller letters in between: five minutes a week on average, and no letter that still greets the customer before.

M04 Receipt Sorter

The chore

Jonas travels to customers twice a month and pays for fuel, trains, hotels and the odd team lunch. The receipts end up as photos on his phone and as PDF files in his mail, and at the end of the month the expense form wants them by category with a total. Finding the twelve files, renaming them, adding up and typing the sums takes an evening that nobody pays for.

What you get

You drop every receipt into one inbox folder, named the way you would name it anyway: date, what it was, the amount. Once a day the Receipt Sorter moves each file into a folder for its month and category, and writes an Excel report per month with the sums by category on the first sheet and every receipt on the second. At the end of the month the report is there, and the receipts are next to it.

Before you start

The setup wizard has run. Make the inbox folder, and name new receipts so that the name holds the date and the amount:

```
2026-10-03 fuel 45.20.pdf
20261005 rail ticket 89,90.pdf
07.10.2026 lunch team 64.50.jpg
```

The date can be written as 2026-10-03, 20261003 or 03.10.2026. The amount is the last number with two decimals, with a point or a comma. A receipt without a date in its name goes into the month of the file's time, which for a photo is the day you took it.

The program

The program reads its settings, moves the receipts and writes the report of every month it touched:

```

' Receipt Sorter: moves the receipts from an inbox folder into one
' folder per month and category, and writes the expense report of
' every month it touched. The settings are the [receipt_sorter] part
' of work.conf.
,
'   jdbasic receipt_sorter.jdb                file the receipts
'   jdbasic receipt_sorter.jdb --dry-run      show, move nothing
'   jdbasic receipt_sorter.jdb --month 2026-10 only the report

IMPORT WORKCONF, RECEIPTS

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "receipt_sorter."
DIM work$ = "~/Documents/AutomateWork/"
DIM inbox$ = WORKCONF.FOLDER$(cfg, key$ + "inbox", _
    work$ + "receipts/inbox")
DIM root$ = WORKCONF.FOLDER$(cfg, key$ + "folder", work$ + "receipts")
DIM kinds = WORKCONF.VALUE(cfg, key$ + "kinds", _
    ["pdf", "jpg", "jpeg", "png"])
DIM rules = RECEIPTS.RULES(WORKCONF.VALUE(cfg, key$ + "rules", _
    ["fuel = Travel", "rail = Travel", "hotel = Travel", _
    "taxi = Travel", "office = Office", "lunch = Meals"]))
DIM dry_run = WORKCONF.FLAG("dry-run")

' Writes the report of one month, or shows where it would go.
SUB Report(root$, month$, dry_run)
    DIM folder$ = PATH.JOIN$(root$, month$)
    DIM rows = RECEIPTS.MONTH_ROWS(folder$)
    DIM path$ = PATH.JOIN$(folder$, month$ + " expenses.xlsx")
    DIM sums = RECEIPTS.SUMS(rows)
    DIM total = sums[LEN(sums) - 1][1]
    DIM shown$ = month$ + ": " + STR$(LEN(rows) - 1) + _
        " receipts, " + FORMAT$("{:.2f}", total)
    IF dry_run THEN
        PRINT "would write the report "; shown$
    ELSE
        RECEIPTS.WRITE_REPORT(path$, rows)
        PRINT "report "; shown$
    ENDIF
ENDSUB

DIM only$ = WORKCONF.ARG$("month")
IF only$ <> "" THEN
    Report(root$, only$, dry_run)
END
ENDIF

IF NOT FILE.ISDIR(inbox$) THEN THROW "No folder " + inbox$

```

```

DIM names = SORT(DIR$(PATH.JOIN$(inbox$, "*")))
DIM months = []
DIM odd = []
DIM k = 0
FOR k = 0 TO LEN(names) - 1
    DIM file$ = names[k]
    DIM from$ = PATH.JOIN$(inbox$, file$)
    DIM ext$ = LCASE$(MID$(PATH.EXT$(file$), 1, 9))
    IF FILE.ISDIR(from$) ORELSE NOT (ext$ IN kinds) THEN
        PUSH(odd, file$)
    ELSE
        DIM info = FILE.STAT(from$)
        DIM r = RECEIPTS.PARSE(file$, rules, info{"mtime"})
        DIM folder$ = PATH.JOIN$(root$, r{"month"}, r{"category"})
        DIM to$ = RECEIPTS.FREE_NAME$(folder$, file$)
        DIM amount$ = "no amount"
        IF r{"amount"} >= 0 THEN _
            amount$ = FORMAT$("{:.2f}", r{"amount"})
        DIM shown$ = file$ + " -> " + r{"month"} + "/" + _
            r{"category"} + " " + amount$
        IF dry_run THEN
            PRINT "would move "; shown$
        ELSE
            MKDIR folder$
            FILE.MOVE(from$, to$)
            PRINT "moved "; shown$
        ENDIF
        IF NOT (r{"month"} IN months) THEN PUSH(months, r{"month"})
    ENDIF
NEXT k

IF LEN(odd) > 0 THEN PRINT "Left in the inbox: "; JOIN(odd, ", ")
IF dry_run THEN END
FOR k = 0 TO LEN(months) - 1
    Report(root$, months[k], FALSE)
NEXT k
IF LEN(names) = 0 THEN PRINT "The inbox is empty."

```

The module RECEIPTS reads names, finds free names and builds the report:

```

' receipts.jdb: receipts filed by month and category. The date and
' the amount come from the file name, such as "2026-10-03 fuel
' 45.20.pdf"; the category from the first rule whose word the name
' contains. A month folder holds one folder per category and the
' expense report.
'
'   IMPORT RECEIPTS

```

```

'   DIM r = RECEIPTS.PARSE("2026-10-03 Shell 45.20.pdf", rules)
'   PRINT r{"month"}, r{"category"}, r{"amount"}

EXPORT MODULE RECEIPTS

IMPORT XLSX

' The rules of the settings, texts like "shell = Travel", as a list
' of maps with "word" in lower case and "category".
EXPORT FUNC RULES(texts)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(texts) - 1
        DIM t$ = texts[k]
        DIM eq = CINT(INSTR(t$, "="))
        IF eq > 0 THEN
            DIM word$ = LCASE$(TRIM$(LEFT$(t$, eq)))
            DIM cat$ = TRIM$(MID$(t$, eq + 1, LEN(t$)))
            IF word$ <> "" ANDALSO cat$ <> "" THEN
                PUSH(out, {"word": word$, "category": cat$})
            ENDIF
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' The category of the first rule whose word is in the name, else
' "Other".
EXPORT FUNC CATEGORY$(name$, rules)
    DIM low$ = LCASE$(name$)
    DIM k = 0
    FOR k = 0 TO LEN(rules) - 1
        DIM r = rules[k]
        IF INSTR(low$, r{"word"}) >= 0 THEN RETURN r{"category"}
    NEXT k
    RETURN "Other"
ENDFUNC

' The date in a file name as YYYY-MM-DD: 2026-10-03, 20261003 or
' 03.10.2026; "" when there is none.
EXPORT FUNC DATE$(name$)
    DIM g = REGEX.FINDALL("(20[0-9]{2})-?([01][0-9])-?([0-3][0-9])", _
        name$)
    IF LEN(g) > 0 THEN RETURN g[0][0] + "-" + g[0][1] + "-" + g[0][2]
    g = REGEX.FINDALL("([0-3][0-9])\.([01][0-9])\. (20[0-9]{2})", _
        name$)
    IF LEN(g) > 0 THEN RETURN g[0][2] + "-" + g[0][1] + "-" + g[0][0]
    RETURN ""

```

```

ENDFUNC

' The amount in a file name: the last number with two decimals, with a
' point or a comma, such as 45.20 or 45,20; -1 when there is none.
EXPORT FUNC AMOUNT(name$)
    DIM stem$ = LEFT$(name$, LEN(name$) - LEN(PATH.EXT$(name$)))
    DIM g = REGEX.FINDALL("([0-9]+)[.,]([0-9]{2})(?![0-9])", stem$)
    IF LEN(g) = 0 THEN RETURN -1
    DIM last = g[LEN(g) - 1]
    RETURN VAL(last[0] + "." + last[1])
ENDFUNC

' Date, month, category and amount of a receipt; changed$ is the
' file's time, used when the name has no date.
EXPORT FUNC PARSE(name$, rules, changed$ = "")
    DIM day$ = DATE$(name$)
    IF day$ = "" THEN day$ = LEFT$(changed$, 10)
    DIM out AS MAP
    out{"date"} = day$
    out{"month"} = LEFT$(day$, 7)
    out{"category"} = CATEGORY$(name$, rules)
    out{"amount"} = AMOUNT(name$)
    RETURN out
ENDFUNC

' A path in the folder for the file name that no file uses yet,
' numbered like "x (2).pdf" when needed.
EXPORT FUNC FREE_NAME$(folder$, file_name$)
    DIM candidate$ = PATH.JOIN$(folder$, file_name$)
    IF NOT FILE.EXISTS(candidate$) THEN RETURN candidate$
    DIM ext$ = PATH.EXT$(file_name$)
    DIM stem$ = LEFT$(file_name$, LEN(file_name$) - LEN(ext$))
    DIM n = 2
    DO
        DIM numbered$ = stem$ + " (" + STR$(n) + ")" + ext$
        candidate$ = PATH.JOIN$(folder$, numbered$)
        n = n + 1
    LOOP WHILE FILE.EXISTS(candidate$)
    RETURN candidate$
ENDFUNC

' The rows of the expense report of a month folder: the header, then
' one row per file in its category folders, by category and name.
EXPORT FUNC MONTH_ROWS(month_folder$)
    DIM rows = [{"Date", "Category", "File", "Amount"}]
    IF NOT FILE.ISDIR(month_folder$) THEN RETURN rows
    DIM cats = SORT(DIR$(PATH.JOIN$(month_folder$, "*")))
    DIM k = 0

```

```

DIM j = 0
FOR k = 0 TO LEN(cats) - 1
    DIM cat$ = cats[k]
    DIM cat_dir$ = PATH.JOIN$(month_folder$, cat$)
    IF FILE.ISDIR(cat_dir$) THEN
        DIM files = SORT(DIR$(PATH.JOIN$(cat_dir$, "*")))
        FOR j = 0 TO LEN(files) - 1
            DIM f$ = files[j]
            DIM amount = AMOUNT(f$)
            DIM day$ = DATE$(f$)
            IF amount >= 0 THEN
                PUSH(rows, [day$, cat$, f$, amount])
            ELSE
                PUSH(rows, [day$, cat$, f$, ""])
            ENDIF
        NEXT j
    ENDIF
NEXT k
RETURN rows
ENDFUNC

' The sum per category of the report rows, and "Total" last.
EXPORT FUNC SUMS(rows)
    DIM out = [{"Category", "Amount", "Receipts"}]
    DIM where AS MAP
    DIM total = 0
    DIM k = 0
    FOR k = 1 TO LEN(rows) - 1
        DIM cat$ = rows[k][1]
        IF NOT MAP.EXISTS(where, cat$) THEN
            where{cat$} = LEN(out)
            PUSH(out, [cat$, 0, 0])
        ENDIF
        DIM sum_row = out[where{cat$}]
        DIM v = rows[k][3]
        IF TYPEOF(v) <> "STRING" THEN
            sum_row[1] = sum_row[1] + v
            total = total + v
        ENDIF
        sum_row[2] = sum_row[2] + 1
    NEXT k
    PUSH(out, ["Total", total, LEN(rows) - 1])
    RETURN out
ENDFUNC

' Writes the expense report: the sums first, then every receipt.
EXPORT SUB WRITE_REPORT(path$, rows)
    DIM wb = XLSX.NEW()

```

```

DIM s = XLSX.SHEET(wb, "Sums", SUMS(rows))
XLSX.COLUMN(s, "A", {"width": 20})
DIM r = XLSX.SHEET(wb, "Receipts", rows)
XLSX.COLUMN(r, "C", {"width": 40})
XLSX.FREEZE(r, 1)
XLSX.WRITE(path$, wb)
ENDSUB

```

How it works

1. **Rules.** The rules setting is a list of texts such as "fuel = Travel". RULES splits each one at the = into a word and a category and drops texts without one. CATEGORY\$ takes the first rule whose word appears anywhere in the file name, in any case; a name no rule fits goes to Other.
2. **Date and amount.** DATE\$ and AMOUNT look for the patterns with REGEX.FINDALL, which answers the groups of each match. The amount takes the last match, because a date such as 03.10 looks like an amount too, and the amount usually comes last in a name.
3. **Choosing the files.** Only files with an extension from kinds are moved. Everything else, such as a text file with notes, stays in the inbox and is named at the end of the run.
4. **Moving.** FREE_NAME\$ checks whether the target name is taken and adds (2), (3) and so on until it is free. No receipt overwrites another, even when you drop the same file twice.
5. **The report.** MONTH_ROWS reads the month folder again, all category folders, and makes one row per file. The report thus always shows what is in the folder, including files you moved there by hand. SUMS adds up per category and counts the receipts; WRITE_REPORT writes both tables into 2026-10 expenses.xlsx.

Run it

```

jdbasic receipt_sorter.jdb --dry-run
would move 07.10.2026 lunch team 64.50.jpg -> 2026-10/Meals 64.50
would move 2026-10-03 fuel 45.20.pdf -> 2026-10/Travel 45.20
would move 20261005 Rail ticket 89,90.pdf -> 2026-10/Travel 89.90
would move office paper 12.99.png -> 2026-10/Office 12.99
Left in the inbox: notes.txt

```

The office paper has no date in its name and goes by the file's time. Then file them:

```
jdbasic receipt_sorter.jdb
moved 07.10.2026 lunch team 64.50.jpg -> 2026-10/Meals 64.50
...
Left in the inbox: notes.txt
report 2026-10: 4 receipts, 212.59
```

To write a month's report again after you changed something by hand:

```
jdbasic receipt_sorter.jdb --month 2026-10
```

Schedule it

The wizard runs the sorter every day at the end of your working day, so the inbox is empty each morning and the report of the month is up to date.

Make it yours

The settings:

```
[receipt_sorter]
inbox = "~/Documents/AutomateWork/receipts/inbox"
folder = "~/Documents/AutomateWork/receipts"
kinds = ["pdf", "jpg", "jpeg", "png", "heic"]
rules = ["fuel = Travel", "rail = Travel", "hotel = Travel",
         "taxi = Travel", "office = Office", "lunch = Meals",
         "parking = Travel"]
```

A new rule is a new line in rules. The first rule that fits wins, so put the narrow ones first: "hotel bar = Meals" before "hotel = Travel".

Changes in the code:

- **The report in another folder**, such as a shared drive of the accounting department: in Report, change path\$ to PATH.JOIN\$("S:/Accounting", month\$ + " Jonas.xlsx").
- **A column for the VAT** in the report: in MONTH_ROWS, add ROUND(amount * 19 / 119 * 100) / 100 as a fifth cell to the row with an amount and "" to the row without, and add "VAT" to the header row.

- **Rename the file** to a clean form while moving it, such as 2026-10-03 Travel 45.20.pdf: in the program, build the new name as `r{"date"} + " " + r{"category"} + " " + amount$ + "." + ext$` and pass it to `FREE_NAME$` instead of `file$`.
- **Mail the report** to the accounting department at the end of the month: build a message with `MAIL.MESSAGE` and `MAIL.ATTACH`, and put it into the outbox following the outbox rule of this chapter; M01 shows the lines.

When it goes wrong

- **A receipt is in the wrong month:** its name has no date and the file's time is the day you saved it, not the day you paid. Put the date at the front of the name.
- **"no amount"** on a receipt: the name has no number with two decimals, such as `fuel 45.pdf`. Write `45.00`.
- **A receipt counts twice in the report:** the same file was dropped twice and filed as `... (2).pdf`. Delete the copy and run with `--month`.
- **A receipt went to Other:** no rule word is in its name. Add a rule, move the file back into the inbox, and run again.

Balance dividend

An evening a month of searching, renaming and adding up, about fifteen minutes a week, and a report that is ready before the form asks for it.

M05 Shift and Vacation Planner

The chore

Jonas leads a team of nine. The plan of who works the early shift, who is on vacation and who is at a training lives in an Excel file on the team drive. Everyone has their own copy of the truth: Ann writes her vacation into Outlook, Ben looks at the sheet once a month, and Cem asks in the chat whether Dora is in on Friday.

Twice a month Jonas finds out too late that three people are away on the same day. Keeping the sheet and a dozen calendars in line costs him about half an hour every week, and the clashes cost more.

What you get

The sheet stays the one place where the plan is made. Every morning the planner reads it and writes a calendar file next to it. Everyone in the team subscribes to that file once, in Outlook, in Google Calendar or on the phone, and from then on sees the team plan beside their own appointments. When the sheet changes, the calendar changes with it.

On every run the planner also checks the plan and names the days on which more people are away than the team can spare.

Before you start

The plan is an Excel file with one sheet and one row per entry. The first row holds the column names, the rest holds the entries:

Name	Kind	From	To	Note
Ann	Vacation	2026-10-19	2026-10-23	
Ben	Vacation	2026-10-20	2026-10-21	
Cem	Training	2026-10-21		Excel course
Dora	Early shift	2026-10-19	2026-10-23	

From and *To* can be real Excel dates or text such as 2026-10-19 or 19.10.2026. An entry without *To* lasts one day. Rows without a name or a start date are left out, so notes below the table do no harm.

The calendar file has to go to a place everyone can reach: a shared folder that is published on the intranet, a SharePoint library or a folder your calendar program can subscribe to. Ask IT which one your office uses; Outlook and Google Calendar both subscribe to an .ics file by its address.

The program

The program reads the settings and the sheet, asks the module for the calendar and the clashes, and writes or shows the result:

```
' Shift and Vacation Planner: turns the team's plan in an Excel sheet
' into a calendar file everyone subscribes to, and warns about days
' on which too many people are away. The settings are the
' [shift_planner] part of work.conf.
'
'   jdbcbasic shift_planner.jdb           write the calendar
'   jdbcbasic shift_planner.jdb --dry-run  show it, write nothing

IMPORT WORKCONF, SHIFTS, XLSX, ICAL

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "shift_planner."
DIM plan$ = WORKCONF.FOLDER$(cfg, key$ + "plan_file", "")
DIM sheet$ = WORKCONF.VALUE(cfg, key$ + "sheet", "Plan")
DIM ics$ = WORKCONF.FOLDER$(cfg, key$ + "calendar_file", "")
DIM title$ = WORKCONF.VALUE(cfg, key$ + "title", "Team plan")
DIM away_kinds = WORKCONF.VALUE(cfg, key$ + "away_kinds", _
    ["Vacation", "Sick", "Training"])
DIM max_away = WORKCONF.VALUE(cfg, key$ + "max_away", 2)
DIM dry_run = WORKCONF.FLAG("dry-run")
IF plan$ = "" ORELSE ics$ = "" THEN
    THROW "Set plan_file and calendar_file in [shift_planner]."
ENDIF

DIM book = XLSX.READ(plan$)
IF NOT MAP.EXISTS(book, sheet$) THEN
    THROW "The plan has no sheet called " + sheet$
ENDIF
DIM entries = SHIFTS.ENTRIES(book{sheet$})
DIM cal = SHIFTS.CALENDAR(entries, title$)
PRINT LEN(entries); " entries in "; PATH.BASENAME$(plan$)
```

```

DIM clashes = SHIFTS.CLASHES(entries, away_kinds, max_away)
DIM k = 0
FOR k = 0 TO LEN(clashes) - 1
    DIM c = clashes[k]
    DIM names$ = JOIN(c{"names"}, ", ")
    PRINT "too many away on "; c{"day"}; ": "; names$
NEXT k

IF dry_run THEN
    FOR k = 0 TO LEN(entries) - 1
        DIM e = entries[k]
        PRINT " "; e{"from"}; " to "; e{"to"}; " "; e{"name"};
        PRINT ": "; e{"kind"}
    NEXT k
    PRINT "Nothing written. Run without --dry-run to write "; ics$
ELSE
    MKDIR PATH.DIRNAME$(ics$)
    ICAL.WRITEFILE(cal, ics$)
    PRINT "wrote "; ics$
ENDIF

```

The module does the work. It turns cells into days, rows into entries, entries into calendar events, and counts who is away on which day:

```

' shifts.jdb: the team's shifts and absences from a sheet, as a
' calendar everyone can subscribe to, and the days on which too many
' people are away.
'
'   IMPORT SHIFTS
'   DIM plan = SHIFTS.ENTRIES(XLSX.READ("plan.xlsx"){"Plan"})
'   ICAL.WRITEFILE(SHIFTS.CALENDAR(plan, "Team"), "team.ics")
'   PRINT SHIFTS.CLASHES(plan, ["Vacation", "Sick"], 2)

EXPORT MODULE SHIFTS

IMPORT DT, ICAL

' A day as YYYY-MM-DD from a cell: an Excel date number (day 1 is
' 1899-12-31), or text written 2026-10-19 or 19.10.2026.
' Answers "" for a cell that holds no date.
EXPORT FUNC DAY$(cell)
    IF TYPEOF(cell) <> "STRING" THEN
        IF cell < 1 THEN RETURN ""
        DIM base = DT.PARSE("1899-12-30")
        RETURN DT.FMT$(base + FLOOR(cell) * 86400, "%Y-%m-%d", 0)
    ENDIF

```

```

    DIM t$ = TRIM$(cell)
    DIM iso = REGEX.FINDALL("^\d{4}-\d{2}-\d{2}$", t$)
    IF LEN(iso) > 0 THEN RETURN t$
    DIM german = REGEX.FINDALL("^\d{1,2}\.\d{1,2}\.\d{4}$", t$)
    IF LEN(german) > 0 THEN
        DIM p = SPLIT(t$, ".")
        DIM dd$ = RIGHT$("0" + p[0], 2)
        DIM mm$ = RIGHT$("0" + p[1], 2)
        RETURN p[2] + "-" + mm$ + "-" + dd$
    ENDIF
    RETURN ""
ENDFUNC

FUNC Cell$(row, i)
    IF i >= LEN(row) THEN RETURN ""
    RETURN TRIM$(" " + row[i])
ENDFUNC

' The rows of the sheet below its header (Name, Kind, From, To,
' Note) as maps with "name", "kind", "from", "to", "note" and "row"
' (the line in the sheet). Rows without a name or a start are left
' out; a row without an end lasts one day.
EXPORT FUNC ENTRIES(rows)
    DIM out = []
    DIM k = 0
    FOR k = 1 TO LEN(rows) - 1
        DIM r = rows[k]
        DIM who$ = Cell$(r, 0)
        DIM first$ = ""
        IF LEN(r) > 2 THEN first$ = DAY$(r[2])
        IF who$ <> "" AND ALSO first$ <> "" THEN
            DIM last$ = first$
            IF LEN(r) > 3 THEN
                DIM d$ = DAY$(r[3])
                IF d$ <> "" THEN last$ = d$
            ENDIF
            PUSH(out, {"name": who$, "kind": Cell$(r, 1), _
                "from": first$, "to": last$, _
                "note": Cell$(r, 4), "row": k + 1})
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' The number of days from the first to the last day, both counted.
EXPORT FUNC DAYS(e)
    DIM a = DT.PARSE(e{"from"})
    DIM b = DT.PARSE(e{"to"})

```

```

    RETURN DT.DIFF(a, b, "days") + 1
ENDFUNC

' A UID that stays the same for the same entry on every build, so a
' subscribed calendar updates instead of doubling its events.
FUNC Uid$(e)
    DIM key$ = e{"name"} + "|" + e{"kind"} + "|" + e{"from"}
    RETURN LEFT$(CODEC.SHA256$(key$), 20) + "@team-plan"
ENDFUNC

' The calendar of all entries: one all-day event each, titled
' "Name: Kind".
EXPORT FUNC CALENDAR(entries, title$)
    DIM cal = ICAL.NEW()
    ICAL.SETPROP(cal, cal, "X-WR-CALNAME", title$)
    DIM k = 0
    FOR k = 0 TO LEN(entries) - 1
        DIM e = entries[k]
        DIM summary$ = e{"name"} + ": " + e{"kind"}
        DIM first = DT.PARSE(e{"from"})
        DIM ev = ICAL.ADDALLDAY(cal, summary$, first, DAYS(e))
        ICAL.SETPROP(cal, ev, "UID", Uid$(e))
        IF e{"note"} <> "" THEN
            ICAL.SETTEXT(cal, ev, "DESCRIPTION", e{"note"})
        ENDIF
    NEXT k
    RETURN cal
ENDFUNC

' The days on which more than max_away people are away, an entry of
' one of the kinds in away_kinds meaning away: maps with "day" and
' "names", sorted by day.
EXPORT FUNC CLASHES(entries, away_kinds, max_away)
    DIM away AS MAP
    DIM low = SELECT(LAMBDA a$ -> LCASE$(a$), away_kinds)
    DIM gone = FILTER(LAMBDA USE(low) e -> LCASE$(e{"kind"}) IN low, _
        entries)
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(gone) - 1
        DIM e = gone[k]
        DIM t = DT.PARSE(e{"from"})
        FOR j = 1 TO DAYS(e)
            DIM day$ = DT.FMT$(t, "%Y-%m-%d", 0)
            DIM who = []
            IF MAP.EXISTS(away, day$) THEN who = away{day$}
            IF NOT (e{"name"} IN who) THEN PUSH(who, e{"name"})
            away{day$} = who
        NEXT j
    NEXT k

```

```

        t = t + 86400
    NEXT j
NEXT k
DIM busy = FILTER(LAMBDA USE(away, max_away) d$ -> _
    LEN(away{d$}) > max_away, SORT(MAP.KEYS(away)))
RETURN SELECT(LAMBDA USE(away) d$ -> _
    {"day": d$, "names": away{d$}}, busy)
ENDFUNC

```

How it works

1. `XLSX.READ` answers every sheet of the workbook as a list of rows, each row a list of cells. Text comes back as text, a date as the number Excel stores for it: the days since the end of 1899.
2. `SHIFTS.DAY$` turns any of those into `YYYY-MM-DD`. A number is added as days to 1899-12-30, which is where Excel's count starts. A text is accepted when it looks like 2026-10-19 or 19.10.2026; the regular expressions in `REGEX.FINDALL` check that. Everything else becomes "", which `ENTRIES` reads as "no date".
3. `SHIFTS.ENTRIES` skips the header row and builds one map per row with a name and a start. It also keeps the row number, so a warning can point to the line in the sheet.
4. `SHIFTS.CALENDAR` makes one all-day event per entry with `ICAL.ADDALLDAY`. The number of days comes from `DT.DIFF`, plus one, because an entry from Monday to Friday covers both days.
5. Each event gets a UID, the name calendar programs use to recognise an event. `Uid$` makes it from the name, the kind and the start with `CODEC.SHA256$`, so the same entry has the same UID on every run. Without that, every subscriber would see the vacation twice after the next update.
6. `SHIFTS.CLASHES` lowercases the kinds that mean away with a `SELECT` and keeps the absences among the entries with a `FILTER`. A loop walks through every day of every absence and collects the names per day in a map. A second `FILTER` keeps the days with more names than `max_away`, a `SELECT` turns each into a map with "day" and "names", and the program prints them.

Run it

With `--dry-run` the planner reads the sheet, prints the warnings and the entries, and writes nothing:

```
jdbasic shift_planner.jdb --dry-run
4 entries in plan.xlsx
too many away on 2026-10-21: Ann, Ben, Cem
  2026-10-19 to 2026-10-23  Ann: Vacation
  2026-10-20 to 2026-10-21  Ben: Vacation
  2026-10-21 to 2026-10-21  Cem: Training
  2026-10-19 to 2026-10-23  Dora: Early shift
Nothing written. Run without --dry-run to write S:/Team/team.ics
```

Without the switch it writes the calendar file and tells you where.

Schedule it

The wizard plans the planner for 07:00 every day, before the team starts. If the sheet changes often during the day, change the schedule to every 2 hours.

Make it yours

The settings are the `[shift_planner]` part of `work.conf`:

```
[shift_planner]
plan_file = "C:/Users/jonas/Team/plan.xlsx"
calendar_file = "S:/Team/Calendar/team.ics"
sheet = "Plan"
title = "Sales team"
away_kinds = ["Vacation", "Sick", "Training"]
max_away = 2
```

Three changes in the code that teams ask for:

- **Early shifts count as away from the office.** Add the kind to `away_kinds` in `work.conf`; no code needed. Kinds match in any case, so early shift in the sheet counts too.
- **A reminder the evening before a vacation.** In `CALENDAR`, after the line with `ICAL.ADDALLDAY`, add an alarm for the people who subscribe:


```
IF LCASE$(e{"kind"}) = "vacation" THEN
  DIM alarm = ICAL.ADDALARM(cal, ev, 12 * 60)
ENDIF
```

The number is minutes before the start; twelve hours before midnight is the noon of the day before.

- **One calendar per person.** In the program, before the line that writes the calendar, filter the entries for a name and write a second file:

```
DIM mine = FILTER(LAMBDA e -> e{"name"} = "Ann", entries)
ICAL.WRITEFILE(SHIFTS.CALENDAR(mine, "Ann"), "ann.ics")
```

When it goes wrong

- **"The plan has no sheet called Plan":** the sheet in your workbook has another name. Set sheet in `work.conf` to the name on its tab.
- **An entry is missing from the calendar:** its row has no name, or a start that is neither an Excel date nor written like 2026-10-19 or 19.10.2026. A date such as Oct 19 is text Excel did not recognise; type it again as a date.
- **Subscribers see old data:** calendar programs fetch a subscribed file on their own timetable, Outlook every few hours, Google up to once a day. The file itself is current after each run.

Balance dividend

About 30 minutes a week of keeping calendars in line, and the clashes found when there is still time to talk about them.

M06 Friday Status Mail

The chore

Every Friday afternoon Jonas writes the status mail for his manager and the team. He scrolls through the chat for what got done, asks three people what they are stuck on, copies it all into a mail and tidies the formatting. It takes 45 minutes on a good Friday, and on a bad one the mail goes out on Monday.

What you get

Each person in the team keeps a small notes file during the week, one line per item with a check box. On Friday afternoon the status mail collects all files and puts the finished mail into the outbox: a summary line, then for every person what is blocked, what is open and what is done. The mail has a plain text part and an HTML part with a table, so it reads well in any mail program.

Jonas opens the mail in the outbox, adds a sentence of his own if he wants, and sends it.

Before you start

The team needs a shared folder with one text file per person, named after them: `Ann.txt`, `Ben.txt`, and so on. In the file, a line that starts with a check box is an item; every other line is ignored, so a heading or a comment does no harm:

```
# Ann, week 43
- [x] Offer for Miller sent
- [ ] Call the printer about the toner
- [!] Contract waits for legal
```

[x] means done, [] still to do, [!] blocked. The x may be a capital letter too. Markdown editors and many note apps show these lines as real check boxes.

The mail server goes into the [mail] part of `work.conf` once, as the introduction of this chapter describes; the status mail follows the outbox rule of this chapter.

The program

The program reads the settings, collects the notes, builds the mail and puts it into the outbox. With `--send` it sends what the outbox holds instead:

```
' Friday Status Mail: collects what the team noted as done, open and
' blocked, and puts the weekly status mail into the outbox. The
' settings are the [friday_status] part of work.conf, the mail server
' the [mail] part.
'
'   jdbasic friday_status.jdb           the mail into the outbox
'   jdbasic friday_status.jdb --dry-run show the mail, change nothing
'   jdbasic friday_status.jdb --send    send what is in the outbox

IMPORT WORKCONF, STATUS, OUTBOX, MAIL

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
    PRINT result{"sent"}; " sent"
    DIM f = 0
    FOR f = 0 TO LEN(result{"failed"}) - 1
        PRINT "   not sent: "; result{"failed"}[f]
    NEXT f
    END
ENDIF

DIM key$ = "friday_status."
DIM here$ = PATH.DIRNAME$(OS.ARGS()[0])
DIM notes$ = WORKCONF.FOLDER$(cfg, key$ + "notes_folder", "")
DIM team = WORKCONF.VALUE(cfg, key$ + "to", [])
DIM subject$ = WORKCONF.VALUE(cfg, key$ + "subject", "Status {week}")
DIM own$ = PATH.JOIN$(here$, "templates", "status.html")
DIM tpl$ = WORKCONF.FOLDER$(cfg, key$ + "template", own$)
DIM clear_done = WORKCONF.VALUE(cfg, key$ + "clear_done", FALSE)
DIM dry_run = WORKCONF.FLAG("dry-run")
IF notes$ = "" THEN THROW "Set notes_folder in [friday_status]."
IF LEN(team) = 0 THEN THROW "Set to in [friday_status]."
IF NOT FILE.EXISTS(tpl$) THEN THROW "No mail template at " + tpl$

DIM people = STATUS.NOTES(notes$)
DIM week$ = STATUS.WEEK$(NOW())
DIM title$ = REPLACE$(subject$, "{week}", week$)
DIM msg = MAIL.MESSAGE(OUTBOX.FROM$(cfg), team, title$)
MAIL.PLAIN(msg, STATUS.PLAIN$(people, week$))
MAIL.HTMLBODY(msg, STATUS.HTML$(people, week$, tpl$))

IF dry_run THEN
```

```

PRINT "To: "; JOIN(team, ", ")
PRINT "Subject: "; title$
PRINT
PRINT STATUS.PLAIN$(people, week$)
PRINT "Nothing written. Run without --dry-run for the outbox."
ELSE
DIM put$ = OUTBOX.PUT$(msg, cfg, "status-" + week$ + ".eml")
PRINT "The status mail is in the outbox: "; put$
IF clear_done THEN
    DIM k = 0
    FOR k = 0 TO LEN(people) - 1
        DIM who$ = people[k>{"person"}
        DIM file$ = PATH.JOIN$(notes$, who$ + ".txt")
        TXTWRITER file$, STATUS.WITHOUT_DONE$(TXTREADER$(file$))
    NEXT k
    PRINT "Done notes removed for next week."
ENDIF
PRINT "Run with --send to send it."
ENDIF

```

The module reads the notes and writes both parts of the mail:

```

' status.jdb: the weekly status of a team from one notes file per
' person, as a plain text and an HTML mail. A note is a check box
' line: "- [x]" done, "- [ ]" still to do, "- [!]" blocked.
'
'   IMPORT STATUS
'   DIM people = STATUS.NOTES("C:/Team/status")
'   PRINT STATUS.PLAIN$(people, "2026-W43")

EXPORT MODULE STATUS

IMPORT TMPL, DT

DIM gLF$ = CHR$(10)

' The notes of one file: a map with "done", "todo" and "blocked",
' each a list of texts in the order of the file.
EXPORT FUNC READ_NOTES(text$)
    DIM out = {"done": [], "todo": [], "blocked": []}
    DIM rows = SPLIT(REPLACE$(text$, CHR$(13), ""), gLF$)
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM t$ = TRIM$(rows[k])
        DIM box$ = LCASE$(LEFT$(t$, 5))
        DIM what$ = TRIM$(MID$(t$, 5, LEN(t$)))
        DIM kind$ = ""

```

```

        IF box$ = "- [x]" THEN kind$ = "done"
        IF box$ = "- [ ]" THEN kind$ = "todo"
        IF box$ = "- [!]" THEN kind$ = "blocked"
        IF kind$ <> "" ANDALSO what$ <> "" THEN
            DIM items = out{kind$}
            PUSH(items, what$)
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' Every person of the notes folder, one per .txt file and named after
' it, sorted by name: maps with "person", "done", "todo", "blocked".
EXPORT FUNC NOTES(folder$)
    DIM out = []
    DIM files = SORT(DIR$(PATH.JOIN$(folder$, "*.txt")))
    DIM k = 0
    FOR k = 0 TO LEN(files) - 1
        DIM file$ = files[k]
        DIM n = READ_NOTES(TXTREADER$(PATH.JOIN$(folder$, file$)))
        DIM ext$ = PATH.EXT$(file$)
        n{"person"} = LEFT$(file$, LEN(file$) - LEN(ext$))
        PUSH(out, n)
    NEXT k
    RETURN out
ENDFUNC

' The ISO week of a day, such as 2026-W43.
EXPORT FUNC WEEK$(t)
    DIM w = DT.ISOWEEK(t)
    RETURN STR$(w{"year"}) + "-W" + RIGHT$("0" + STR$(w{"week"}), 2)
ENDFUNC

' How many notes of a kind all people have together.
EXPORT FUNC TOTAL(people, kind$)
    RETURN SUM(SELECT(LAMBDA USE(kind$) p -> LEN(p{kind$}), people))
ENDFUNC

SUB AddList(out, title$, items)
    IF LEN(items) = 0 THEN RETURN
    PUSH(out, " " + title$)
    DIM k = 0
    FOR k = 0 TO LEN(items) - 1
        PUSH(out, "    - " + items[k])
    NEXT k
ENDSUB

' The mail as plain text: a summary line, then each person with the

```

```

' blocked notes first, the open ones next and the done ones last.
EXPORT FUNC PLAIN$(people, week$)
  DIM out = ["Status " + week$, ""]
  DIM sum$ = STR$(TOTAL(people, "done")) + " done, "
  sum$ = sum$ + STR$(TOTAL(people, "todo")) + " open, "
  PUSH(out, sum$ + STR$(TOTAL(people, "blocked")) + " blocked.")
  DIM k = 0
  FOR k = 0 TO LEN(people) - 1
    DIM p = people[k]
    PUSH(out, "")
    PUSH(out, p{"person"})
    AddList(out, "Blocked:", p{"blocked"})
    AddList(out, "Open:", p{"todo"})
    AddList(out, "Done:", p{"done"})
  NEXT k
  RETURN JOIN(out, gLF$) + gLF$
ENDFUNC

' The mail as HTML from a TMPL template that sees "week", "people"
' and the totals "done", "todo" and "blocked".
EXPORT FUNC HTML$(people, week$, template$)
  DIM m AS MAP
  m{"week"} = week$
  m{"people"} = people
  m{"done"} = TOTAL(people, "done")
  m{"todo"} = TOTAL(people, "todo")
  m{"blocked"} = TOTAL(people, "blocked")
  RETURN TMPL.RENDER$(template$, m)
ENDFUNC

' The notes with the done ones taken out, for the next week.
EXPORT FUNC WITHOUT_DONE$(text$)
  DIM rows = SPLIT(REPLACE$(text$, CHR$(13), ""), gLF$)
  DIM kept = FILTER(LAMBDA r$ -> _
    LCASE$(LEFT$(TRIM$(r$), 5)) <> "- [x]", rows)
  RETURN JOIN(kept, gLF$)
ENDFUNC

```

The HTML part comes from a template file next to the program, in `templates/status.html`. It is plain HTML with a few marks of the TMPL library: `{{ week }}` is replaced by the week, and `{% for p in people %}` repeats a table row for every person.

How it works

1. STATUS.NOTES lists the .txt files of the folder with DIR\$, sorted by name, reads each with TXTREADER\$, and names the person after the file without its extension.
2. READ_NOTES looks at the first five characters of every line. If they are one of the three check boxes, the rest of the line goes into the list for done, to do or blocked. LCASE\$ makes [X] count as [x].
3. WEEK\$ names the week the way calendars do, with DT.ISOWEEK: the ISO week starts on Monday, and the first of January can belong to the last week of the year before.
4. PLAIN\$ builds the text part line by line into a list and joins it at the end. Blocked items come first, because they are the ones someone has to act on.
5. HTML\$ hands a map with the week, the people and the totals to TMPL.RENDER\$, which fills the template. TOTAL counts the notes of one kind with SUM over a SELECT of the list lengths. TMPL writes every value escaped for HTML, so an item such as *Miller & Co* cannot break the table.
6. MAIL.MESSAGE creates the message with the sender from the [mail] part, and OUTBOX.PUT\$ writes it as status-2026-W43.eml into the outbox. The same week always gives the same file name, so running the program twice replaces the first mail instead of adding a second one.
7. With clear_done = true, the program then rewrites every notes file without its done lines, ready for the next week. WITHOUT_DONE\$ keeps every other line as it was, with a FILTER on the first five characters.

Run it

A dry run prints the mail and changes nothing:

```
jdbasic friday_status.jdb --dry-run
To: team@example.com
Subject: Status 2026-W43

Status 2026-W43

2 done, 2 open, 1 blocked.

Ann
Blocked:
```

```

    - Contract waits for legal
Open:
    - Call the printer about the toner
Done:
    - Offer for Miller sent

Ben
Open:
    - Plan the team day
Done:
    - Budget 2027 draft

Nothing written. Run without --dry-run for the outbox.

```

Without the switch, the mail goes into the outbox, and `jdbasic friday_status.jdb --send` sends it after asking for the password.

Schedule it

The wizard plans the status mail for Fridays at 15:00, which leaves an hour to look at it before the weekend. The `--send` step stays a step you take yourself.

Make it yours

The settings are the `[friday_status]` part of `work.conf`:

```

[friday_status]
notes_folder = "S:/Team/status"
to = ["team@example.com", "boss@example.com"]
subject = "Sales team, status {week}"
clear_done = true

```

Changes in the code:

- **Only the blocked and open items.** In `PLAIN$`, delete the line `AddList(out, "Done:", p{"done"})`. In the template, delete the cell with `p.done` and its column heading.
- **A fourth kind of note**, such as `[?]` for questions to the manager. In `READ_NOTES`, add the kind to the map at the top, `{"done": [], "todo": [], "blocked": [], "ask": []}`, and one line for the box:


```
IF box$ = "- [?]" THEN kind$ = "ask"
```

Then add `AddList(out, "Questions:", p{"ask"})` to `PLAIN$`.

- **People who sent nothing this week.** In `PLAIN$`, after the line `PUSH(out, p{"person"})`, add a line when all three lists are empty:

```
DIM n = LEN(p{"done"}) + LEN(p{"todo"}) + LEN(p{"blocked"})
IF n = 0 THEN PUSH(out, " (no notes this week)")
```

When it goes wrong

- **"Set to in [friday_status]":** the list of recipients is empty. Add at least one address.
- **A person is missing:** the file does not end in `.txt`, or it lies in a subfolder. The program reads the files of the folder itself.
- **An item is missing:** its line does not start with one of the three boxes. A space inside the brackets, `[ x ]`, does not count.
- **"No mail template at":** the templates folder was not copied with the program. Run the wizard again, or copy the folder by hand.

Balance dividend

About 45 minutes every Friday, and a status that goes out on time because it no longer depends on your memory of the week.

M07 Inbox Unpacker

The chore

Invoices, offers, signed contracts and spreadsheets reach Jonas as attachments. To keep them, he opens each mail, saves each file, finds the right folder, and renames `document.pdf` to something he will recognise in March. Half the time he skips it, and later he searches his inbox for "that PDF from Anna in October".

What you get

Jonas saves the mails he wants to keep into one folder. The unpacker takes every saved mail apart: each attachment goes into a folder named after the sender, and inside it into a folder named after the day and the subject:

```
Anna Miller\2026-10-23 Invoice October\invoice 2026-118.pdf
Ben Ortiz\2026-10-26 Re Budget 2027\budget.xlsx
```

The mails themselves move into a done folder, so the next run only looks at new ones. Logos and pictures that sit inside the text of a mail are left out; they are rarely worth keeping.

Before you start

The unpacker reads mails saved as `.eml` files. Most mail programs save these: in Thunderbird, choose *Save As* on a mail; in Outlook on the web and the new Outlook, choose *Download* or *Save as*; in Apple Mail, drag the mail into a Finder folder. The classic Outlook for Windows saves `.msg` files instead, which this recipe cannot read. There, forward the mail to yourself as an attachment and save that, or use Outlook on the web.

Make one folder for the saved mails and one for the attachments. The wizard suggests saved mail and attachments in your `AutomateWork` folder.

The program

The program reads the settings, plans what to save and saves it:

```

' Inbox Unpacker: files the attachments of saved mails into folders by
' sender and subject, then moves the mails into a done folder. The
' settings are the [inbox_unpacker] part of work.conf.
,
'   jdbasic inbox_unpacker.jdb           save and move
'   jdbasic inbox_unpacker.jdb --dry-run  show the plan only

IMPORT WORKCONF, UNPACKER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "inbox_unpacker."
DIM home$ = "~/Documents/AutomateWork/"
DIM inbox$ = WORKCONF.FOLDER$(cfg, key$ + "inbox_folder", _
    home$ + "saved mail")
DIM target$ = WORKCONF.FOLDER$(cfg, key$ + "target_folder", _
    home$ + "attachments")
DIM done$ = WORKCONF.FOLDER$(cfg, key$ + "done_folder", _
    PATH.JOIN$(inbox$, "done"))
DIM keep_inline = WORKCONF.VALUE(cfg, key$ + "keep_inline", FALSE)
DIM dry_run = WORKCONF.FLAG("dry-run")
IF NOT FILE.ISDIR(inbox$) THEN THROW "No mail folder at " + inbox$

DIM plan = UNPACKER.PLAN(inbox$, target$, keep_inline)
DIM mails = LEN(plan{"mails"})
IF mails = 0 THEN
    PRINT "No .eml files in "; inbox$
    END
ENDIF
PRINT mails; " mails in "; inbox$
DIM saved = UNPACKER.APPLY(plan, dry_run, done$)
IF dry_run THEN
    PRINT "Nothing written. Run without --dry-run to save "; saved; _
        " files."
ELSE
    PRINT saved; " files saved, the mails moved to "; done$
ENDIF

```

The module does the work:

```

' unpacker.jdb: the attachments of saved mails (.eml files) filed into
' folders by sender, then by day and subject:
' Anna Miller/2026-10-23 Invoice October/invoice.pdf.
,
'   IMPORT UNPACKER
'   DIM plan = UNPACKER.PLAN("C:/Mail/saved", "C:/Mail/files", FALSE)
'   PRINT UNPACKER.APPLY(plan, FALSE, "C:/Mail/saved/done")

```

```

EXPORT MODULE UNPACKER

IMPORT MAIL, DT

' A text that can be a file or folder name on Windows: the characters
' Windows forbids become spaces, runs of spaces one, and it is cut to
' max_chars characters.
EXPORT FUNC SAFE$(text$, max_chars)
    DIM bad = ["\", "/", ":", "*", "?", CHR$(34), "<", ">", "|"]
    DIM t$ = REDUCE(LAMBDA acc$, b$ -> REPLACE$(acc$, b$, " "), bad, _
        text$)
    t$ = REGEX.REPLACE("[\x00-\x1f]", t$, " ")
    t$ = TRIM$(REGEX.REPLACE(" +", t$, " "))
    DIM chars = -t$
    IF LEN(chars) > max_chars THEN
        t$ = TRIM$(JOIN(TAKE(max_chars, chars), ""))
    ENDIF
    DO WHILE RIGHT$(t$, 1) = "."
        t$ = LEFT$(t$, LEN(t$) - 1)
    LOOP
    IF t$ = "" THEN t$ = "unnamed"
    RETURN t$
ENDFUNC

' The sender's name, or the bare address when the mail has no name.
EXPORT FUNC SENDER$(parsed)
    DIM from$ = parsed{"from"}
    DIM who$ = MAIL.ADDRESSNAME$(from$)
    IF who$ = "" THEN who$ = MAIL.BAREADDRESS$(from$)
    IF who$ = "" THEN who$ = "unknown sender"
    RETURN SAFE$(who$, 50)
ENDFUNC

' The sending day of the mail as YYYY-MM-DD in the offset given, or
' "undated" for a mail without a readable date.
EXPORT FUNC DAY$(parsed, offset)
    DIM t = DT.PARSE(parsed{"date"})
    IF TYPEOF(t) = "NONE" THEN RETURN "undated"
    RETURN DT.FMT$(t, "%Y-%m-%d", offset)
ENDFUNC

' The folder for one mail: sender, then day and subject.
EXPORT FUNC FOLDER$(parsed, target$, offset)
    DIM subject$ = parsed{"subject"}
    IF TRIM$(subject$) = "" THEN subject$ = "no subject"
    DIM inner$ = DAY$(parsed, offset) + " " + SAFE$(subject$, 60)
    RETURN PATH.JOIN$(target$, SENDER$(parsed), inner$)
ENDFUNC

```

```

FUNC FreeName$(folder$, file$, taken)
    DIM candidate$ = PATH.JOIN$(folder$, file$)
    DIM used = FILE.EXISTS(candidate$) ORELSE candidate$ IN taken
    IF NOT used THEN RETURN candidate$
    DIM ext$ = PATH.EXT$(file$)
    DIM stem$ = LEFT$(file$, LEN(file$) - LEN(ext$))
    DIM n = 2
    DO
        DIM alt$ = stem$ + " (" + STR$(n) + ")" + ext$
        candidate$ = PATH.JOIN$(folder$, alt$)
        n = n + 1
    LOOP WHILE FILE.EXISTS(candidate$) ORELSE candidate$ IN taken
    RETURN candidate$
ENDFUNC

' What to do with the .eml files of the inbox folder: a map with
' "files" (one map per attachment with "eml", "index", "to" and
' "short", the path below the target folder) and "mails" (every mail
' looked at). Pictures shown inside the text of a mail (with a
' Content-ID) are left out unless keep_inline is TRUE.
EXPORT FUNC PLAN(inbox$, target$, keep_inline)
    DIM files = []
    DIM mails = []
    DIM taken = []
    DIM offset = DT.LOCAL_OFFSET()
    DIM names = SORT(DIR$(PATH.JOIN$(inbox$, "*.eml")))
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(names) - 1
        DIM base$ = names[k]
        DIM eml$ = PATH.JOIN$(inbox$, base$)
        DIM parsed = MAIL.PARSEFILE(eml$)
        PUSH(mails, eml$)
        DIM folder$ = FOLDER$(parsed, target$, offset)
        DIM att = parsed{"att_names"}
        DIM cids = parsed{"att_cids"}
        FOR j = 0 TO LEN(att) - 1
            DIM cid$ = cids[j]
            IF keep_inline ORELSE cid$ = "" THEN
                DIM att$ = att[j]
                DIM file$ = SAFE$(att$, 80)
                DIM to$ = FreeName$(folder$, file$, taken)
                PUSH(taken, to$)
                DIM short$ = MID$(to$, LEN(target$) + 1, LEN(to$))
                PUSH(files, {"eml": eml$, "index": j, "to": to$, _
                    "short": short$})
            ENDIF
        NEXT j
    NEXT k

```

```

        NEXT j
    NEXT k
    RETURN {"files": files, "mails": mails}
ENDFUNC

' Saves the attachments of the plan and moves every mail it looked at
' into done_folder$; with dry_run it only prints. Answers the number
' of attachments.
EXPORT FUNC APPLY(plan, dry_run, done_folder$)
    DIM files = plan{"files"}
    DIM k = 0
    DIM last$ = ""
    DIM parsed = {}
    FOR k = 0 TO LEN(files) - 1
        DIM f = files[k]
        DIM to$ = f{"to"}
        IF dry_run THEN
            PRINT "would save "; f{"short"}
        ELSE
            IF f{"eml"} <> last$ THEN
                parsed = MAIL.PARSEFILE(f{"eml"})
                last$ = f{"eml"}
            ENDIF
            MKDIR PATH.DIRNAME$(to$)
            MAIL.SAVEATTACHMENT(parsed, f{"index"}, to$)
            PRINT "saved "; f{"short"}
        ENDIF
    NEXT k
    IF NOT dry_run THEN
        MKDIR done_folder$
        DIM mails = plan{"mails"}
        FOR k = 0 TO LEN(mails) - 1
            DIM eml$ = mails[k]
            DIM base$ = PATH.BASENAME$(eml$)
            DIM dest$ = FreeName$(done_folder$, base$, [])
            FILE.MOVE(eml$, dest$)
        NEXT k
    ENDIF
    RETURN LEN(files)
ENDFUNC

```

How it works

1. PLAN lists the .eml files with DIR\$ and reads each one with MAIL.PARSEFILE. The result is a map with the sender, the subject, the date, the text and the

attachments. The names of the attachments are in `att_names`, in the order they appear in the mail.

2. `SENDER$` takes the name from the `From` line with `MAIL.ADDRESSNAME$`. A mail from a bare address such as `bob@supplier.example` has no name, so the address is used.
3. `DAY$` reads the `Date` line with `DT.PARSE`, which understands the format mails use, and writes the day in your own time zone. A mail without a readable date goes into an undated folder.
4. `SAFE$` makes any text usable as a file name. Windows does not allow `\ / : * ? " < > |` in names, and a subject such as *Invoice: October/2026* contains two of them. They become spaces: `REDUCE` hands the text from one `REPLACE$` to the next, once for each forbidden character. `-t$` splits the text into characters, so the cut to 60 characters never breaks a letter such as *ü* in half.
5. A picture inside the text of a mail carries a Content-ID, which `att_cids` lists. Attachments with a Content-ID are left out unless `keep_inline` is on.
6. `FreeName$` makes sure no file is overwritten. When the folder already has an `invoice.pdf`, or the same plan already uses the name, the new file becomes `invoice (2).pdf`.
7. `APPLY` writes each attachment with `MAIL.SAVEATTACHMENT`, which decodes it byte for byte, and then moves every mail it looked at into the done folder with `FILE.MOVE`.

Run it

A dry run shows where every file would go:

```
jdbasic inbox_unpacker.jdb --dry-run
2 mails in C:/Users/Jonas/Documents/AutomateWork/saved mail
would save Anna Miller\2026-10-23 Invoice October\invoice 2026-118.pdf
would save Ben Ortiz\2026-10-26 Re Budget 2027\budget.xlsx
would save Ben Ortiz\2026-10-26 Re Budget 2027\budget notes.docx
Nothing written. Run without --dry-run to save 3 files.
```

Without the switch the files are saved and the mails move into `saved mail\done`. Nothing is ever deleted: if a folder looks wrong, move the mails back out of `done` and run it again.

Schedule it

Once a day is enough; the wizard plans it for 12:30. Jonas saves mails whenever he likes, and after lunch they are filed.

Make it yours

The settings are the [inbox_unpacker] part of work.conf:

```
[inbox_unpacker]
inbox_folder = "D:/Mail/saved"
target_folder = "S:/Team/Documents/from mail"
done_folder = "D:/Mail/saved/done"
keep_inline = false
```

Changes in the code:

- **One folder per month instead of per day.** In DAY\$, write the month only: "%Y-%m" instead of "%Y-%m-%d".
- **One folder per company.** In SENDER\$, use the part of the address after the @. Replace the two lines that set who\$ from the name and the address with:

```
DIM addr$ = MAIL.BAREADDRESS$(from$)
DIM who$ = MID$(addr$, INSTR(addr$, "@") + 1, LEN(addr$))
```

Every mail from miller.example then lands in one folder.

- **Skip calendar invitations and signatures.** In PLAN, add a list of extensions above the loops:

```
DIM skip = [".ics", ".vcf", ".p7s"]
```

In the inner loop, replace the line with IF keep_inline by these three, and delete the DIM att\$ line just below them:

```
DIM att$ = att[j]
DIM wanted = NOT (LCASE$(PATH.EXT$(att$)) IN skip)
IF (keep_inline OR ELSE cid$ = "") AND ALSO wanted THEN
```


When it goes wrong

- **"No mail folder at":** the `inbox_folder` setting points to a folder that does not exist. Check the spelling in `work.conf`.
- **"No .eml files":** the mails were saved as `.msg`, or into a different folder. See *Before you start*.
- **A file named** `unnamed`: the attachment had no name, or only characters Windows does not allow.
- **The day is one off:** the folder uses the day in your time zone, so a mail sent late at night from another continent can land on the next or the previous day.

Balance dividend

About 30 minutes a week of saving and renaming, and every attachment can be found by who sent it and when.

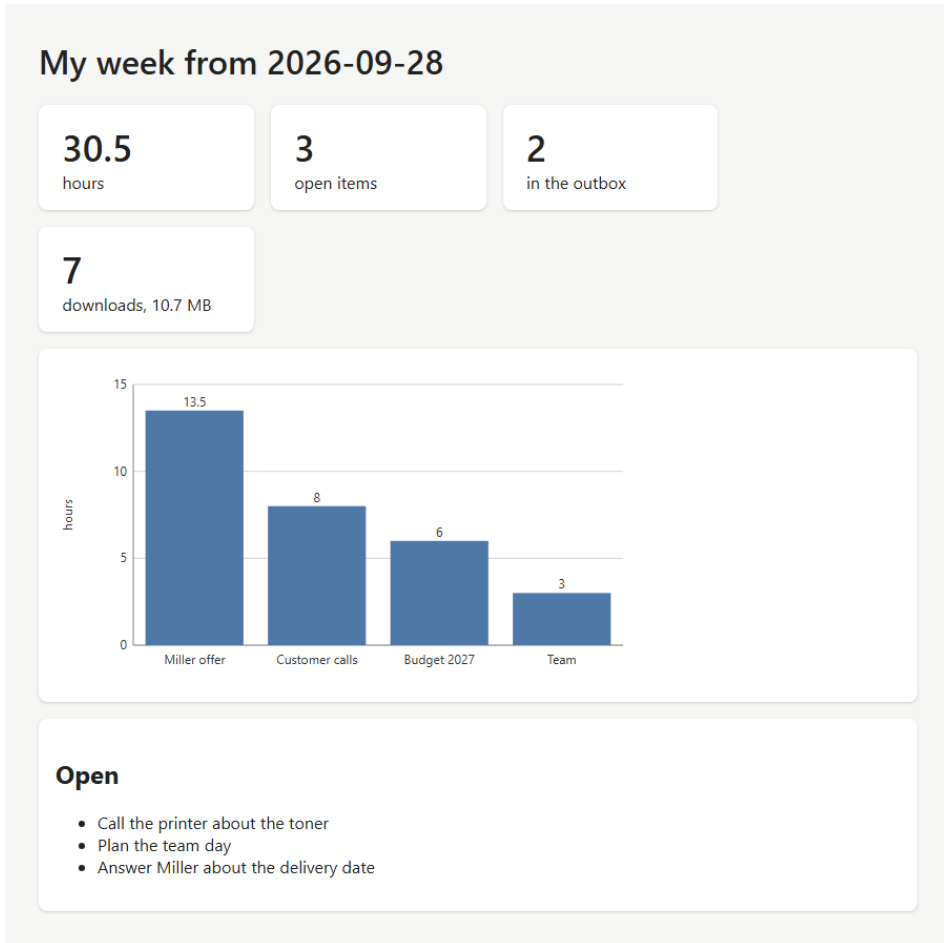
M08 Personal Dashboard

The chore

On Friday afternoon Jonas wants to know where his week went. The hours are in the time tracker's log, the open items in his notes file, the mails he still has to send in the outbox, and the downloads folder fills up without him noticing. Finding each number means opening four places, so he rarely looks.

What you get

A web page on your own computer that shows all of it at once: the hours of this week, a bar chart of the hours per project, the number of open items with the list below, the mails waiting in the outbox, and how many files lie in the downloads folder and how much space they take. The page refreshes itself every five minutes.



The dashboard on a Friday morning

The page is served by a small web server inside the program. It listens only on this computer, at `localhost`, so nobody else on the network can open it, and nothing goes to the internet.

Before you start

The dashboard reads what other recipes write:

- the log of the time tracker (E07), from the `[time_tracker]` part of `work.conf`;
- a notes file with open items written - `[ ] item`, the same check boxes as in the Friday status mail (M06);

- the outbox from the [mail] part, as the outbox rule of this chapter describes.

None of them is required. A missing log shows zero hours, a missing notes file no open items.

The program

The program reads the settings and either serves the page or saves it once as a file:

```
' Personal Dashboard: your numbers of the week on a web page on this
' computer: hours from the time tracker, open items, the outbox and
' the downloads folder. The settings are the [dashboard] part of
' work.conf; the time log is the one of [time_tracker].
'
' jdbasic personal_dashboard.jdb          serve until Ctrl+C
' jdbasic personal_dashboard.jdb --save my.html write the page once

IMPORT WORKCONF, DASHBOARD, JDWEB

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "dashboard."
DIM here$ = PATH.DIRNAME$(OS.ARGS()[0])
DIM time_home$ = "~/Documents/AutomateWork/time"
DIM log_folder$ = WORKCONF.FOLDER$(cfg, "time_tracker.folder", _
    time_home$)
DIM todo$ = WORKCONF.FOLDER$(cfg, key$ + "todo_file", "")
DIM down$ = WORKCONF.FOLDER$(cfg, key$ + "downloads", "~/Downloads")
DIM port = WORKCONF.VALUE(cfg, key$ + "port", 8765)
DIM own$ = PATH.JOIN$(here$, "templates", "dashboard.html")
DIM tpl$ = WORKCONF.FOLDER$(cfg, key$ + "template", own$)
IF NOT FILE.EXISTS(tpl$) THEN THROW "No page template at " + tpl$

DIM log_file$ = PATH.JOIN$(log_folder$, "log.csv")
DIM sources = {"log": log_file$, "todo": todo$, _
    "downloads": down$, "cfg": cfg}

DIM save$ = WORKCONF.ARG$("save")
IF save$ <> "" THEN
    DIM now$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M:%S")
    DIM numbers = DASHBOARD.NUMBERS(sources, now$)
    TXTWRITER save$, DASHBOARD.PAGE$(numbers, tpl$)
    PRINT "wrote "; save$
END
ENDIF
```

```

DASHBOARD.MOUNT(sources, tpl$)
IF NOT JDWEB.LISTEN(port) THEN
    THROW "Port " + STR$(port) + " is taken. Set port in [dashboard]."
ENDIF
PRINT "Your dashboard: http://localhost:"; port; "/"
PRINT "Only this computer can open it. Stop it with Ctrl+C."
HTTP.SERVER.WAIT

```

The module collects the numbers and answers the browser:

```

' dashboard.jdb: your numbers on one local web page: hours this week
' from the time tracker's log, open items, mails waiting in the
' outbox, and what lies in the downloads folder.
'
'   IMPORT DASHBOARD
'   DIM n = DASHBOARD.NUMBERS(sources, "2026-10-23 16:00:00")
'   DASHBOARD.MOUNT(sources, "templates/dashboard.html")
'   JDWEB.SERVE(8765)

EXPORT MODULE DASHBOARD

IMPORT JDWEB, SVG, TMPL, OUTBOX

DIM gSources AS MAP
DIM gTemplate$ = ""

' The Monday of the week a day (YYYY-MM-DD) belongs to.
EXPORT FUNC MONDAY$(day$)
    DIM back = (VAL(FORMAT_DATE(CDATE(day$), "%w")) + 6) MOD 7
    RETURN FORMAT_DATE(DATEADD("D", -back, CDATE(day$)), "%Y-%m-%d")
ENDFUNC

SUB AddBlock(per, monday$, start$, stop$, project$)
    DIM col = DATEDIFF("D", CDATE(monday$), CDATE(start$))
    IF col < 0 ORELSE col > 6 THEN RETURN
    DIM minutes = DATEDIFF("N", CDATE(start$), CDATE(stop$))
    DIM had = 0
    IF MAP.EXISTS(per, project$) THEN had = per{project$}
    per{project$} = had + minutes / 60
ENDSUB

' Hours per project from the time log (time,event,project) for the
' week from monday$: maps with "project" and "hours", most first. A
' block counts on the day it started; a running block until now$.
EXPORT FUNC HOURS(log_path$, monday$, now$)
    DIM per AS MAP

```

```

DIM rows = []
IF FILE.EXISTS(log_path$) THEN
    DIM kinds = ["STRING", "STRING", "STRING"]
    rows = CSVREADER(log_path$, ",", TRUE, kinds)
ENDIF
DIM start$ = ""
DIM project$ = ""
DIM k = 0
FOR k = 0 TO LEN(rows) - 1
    DIM r = rows[k]
    DIM when$ = r[0]
    DIM what$ = r[1]
    IF start$ <> "" THEN
        AddBlock(per, monday$, start$, when$, project$)
    ENDIF
    start$ = ""
    IF what$ = "start" THEN
        start$ = when$
        project$ = ""
        IF LEN(r) > 2 THEN project$ = r[2]
    ENDIF
NEXT k
IF start$ <> "" THEN
    AddBlock(per, monday$, start$, now$, project$)
ENDIF
DIM names = MAP.KEYS(per)
DIM totals = SELECT(LAMBDA USE(per) p$ -> _
    ROUND(per{p$} * 10) / 10, names)
RETURN SELECT(LAMBDA USE(names, totals) i -> _
    {"project": names[i], "hours": totals[i]}, _
    REVERSE(GRADE(totals)))
ENDFUNC

' The open items (lines starting "- [ ]") of a notes file.
EXPORT FUNC TODO(file$)
    IF NOT FILE.EXISTS(file$) THEN RETURN []
    DIM lines = SPLIT(TXTREADER$(file$), CHR$(10))
    DIM open = FILTER(LAMBDA l$ -> LEFT$(TRIM$(l$), 5) = "- [ ]", _
        lines)
    RETURN SELECT(LAMBDA l$ -> TRIM$(MID$(TRIM$(l$), 5, LEN(l$))), _
        open)
ENDFUNC

' The files directly in a folder: a map with "files" and "bytes".
EXPORT FUNC FOLDER(folder$)
    DIM files = 0
    DIM bytes = 0
    IF FILE.ISDIR(folder$) THEN

```

```

    DIM names = DIR$(PATH.JOIN$(folder$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM base$ = names[k]
        DIM info = FILE.STAT(PATH.JOIN$(folder$, base$))
        IF NOT info{"is_dir"} THEN
            files = files + 1
            bytes = bytes + info{"size"}
        ENDIF
    NEXT k
ENDIF
RETURN {"files": files, "bytes": bytes}
ENDFUNC

' A size in bytes as text with KB, MB or GB.
EXPORT FUNC HUMAN$(bytes)
    IF bytes < 1024 THEN RETURN STR$(bytes) + " bytes"
    DIM units = ["KB", "MB", "GB", "TB"]
    DIM v = bytes / 1024
    DIM u = 0
    DO WHILE v >= 1024 AND ALSO u < 3
        v = v / 1024
        u = u + 1
    LOOP
    RETURN STR$(ROUND(v * 10) / 10) + " " + units[u]
ENDFUNC

' All numbers of the page as one map. sources has "log", "todo",
' "downloads" and "cfg" (the settings, for the outbox).
EXPORT FUNC NUMBERS(sources, now$)
    DIM monday$ = MONDAY$(LEFT$(now$, 10))
    DIM hours = HOURS(sources{"log"}, monday$, now$)
    DIM total = SUM(SELECT(LAMBDA h -> h{"hours"}, hours))
    DIM down = FOLDER(sources{"downloads"})
    DIM out = {"week_of": monday$, "hours": hours}
    out{"total_hours"} = ROUND(total * 10) / 10
    out{"todo"} = TODO(sources{"todo"})
    out{"outbox"} = LEN(OUTBOX.PENDING(sources{"cfg"}))
    out{"downloads"} = down{"files"}
    out{"downloads_size"} = HUMAN$(down{"bytes"})
    RETURN out
ENDFUNC

' The hours as a bar chart, SVG markup.
EXPORT FUNC CHART$(numbers)
    DIM hours = numbers{"hours"}
    DIM labels = SELECT(LAMBDA h -> h{"project"}, hours)
    DIM values = SELECT(LAMBDA h -> h{"hours"}, hours)

```

```

    IF LEN(labels) = 0 THEN
        labels = ["no hours yet"]
        values = [0]
    ENDIF
    DIM c = SVG.CHART("bar", 560, 300)
    SVG.SETOPT(c, "legend", "none")
    SVG.SETOPT(c, "values", 1)
    SVG.SETOPT(c, "y_title", "hours")
    SVG.LABELS(c, labels)
    SVG.SERIES(c, "hours", values)
    RETURN SVG.MARKUP$(SVG.RENDER(c))
ENDFUNC

' The whole page from the template.
EXPORT FUNC PAGE$(numbers, template$)
    DIM model = numbers
    model{"chart"} = CHART$(numbers)
    model{"open_count"} = LEN(numbers{"todo"})
    RETURN TMPL.RENDER$(template$, model)
ENDFUNC

FUNC Now$()
    RETURN FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M:%S")
ENDFUNC

FUNC ShowPage(request)
    RETURN PAGE$(NUMBERS(gSources, Now$()), gTemplate$)
ENDFUNC

FUNC ShowData(request)
    RETURN NUMBERS(gSources, Now$())
ENDFUNC

FUNC ShowChart(request)
    DIM svg$ = CHART$(NUMBERS(gSources, Now$()))
    RETURN JDWEB.REPLY(svg$, 200, "image/svg+xml")
ENDFUNC

' Registers the page at /, its numbers at /data.json and the chart
' at /chart.svg.
EXPORT SUB MOUNT(sources, template$)
    gSources = sources
    gTemplate$ = template$
    JDWEB.GET("/", ShowPage@)
    JDWEB.GET("/data.json", ShowData@)
    JDWEB.GET("/chart.svg", ShowChart@)
ENDSUB

```


The look of the page comes from `templates/dashboard.html` next to the program: HTML with a style sheet and the TMPL marks for the numbers.

How it works

1. `MONDAY$` finds the Monday of the current week. `FORMAT_DATE` with `%w` gives the day of the week, 0 for Sunday, and `DATEADD` goes back that many days.
2. `HOURS` reads the log with `CSVREADER`. Every line closes the block that runs and a start line opens a new one, so the minutes between two lines belong to the project of the first. `AddBlock` adds them up per project, but only for blocks that started this week. A block still running counts until now.
3. A `SELECT` rounds the total of each project. `GRADE` answers the positions of the totals from small to large; `REVERSE` turns that around, so the project with the most hours comes first. A second `SELECT` turns each position into a map with project and hours.
4. `TODO` keeps the lines that start with an empty box with a `FILTER` and cuts the box off with a `SELECT`. `FOLDER` adds up the files of the downloads folder with `FILE.STAT`. `LEN(OUTBOX.PENDING(cfg))` is the number of mails waiting.
5. `CHART$` takes the names and the hours with two `SELECTs` and draws the bar chart with `SVG.CHART`, `SVG.LABELS` and `SVG.SERIES`. `SVG` is a picture written as text, so it goes straight into the page.
6. `MOUNT` tells `JDWEB` which function answers which address: `/` for the page, `/data.json` for the numbers alone, `/chart.svg` for the chart. A map returned by a handler goes out as JSON.
7. `JDWEB.LISTEN` opens the port, and `HTTP.SERVER.WAIT` answers the browser until you stop the program. Every visit reads the files again, so the page is always current.

Run it

Start the program and open the address it prints:

```
jdbasic personal_dashboard.jdb
Your dashboard: http://localhost:8765/
Only this computer can open it. Stop it with Ctrl+C.
```

The window stays open while the page is served. To keep a copy of the page without a server, for a mail or for the archive, save it:

```
jdbasic personal_dashboard.jdb --save "my week.html"  
wrote my week.html
```

Schedule it

The wizard starts the dashboard when you log on, so the page is there all day. Put `http://localhost:8765/` into your browser's bookmarks or make it the start page.

Make it yours

The settings are the [dashboard] part of `work.conf`:

```
[dashboard]  
todo_file = "D:/Notes/Jonas.txt"  
downloads = "~/Downloads"  
port = 8765
```

If another program already uses port 8765, choose any number from 1024 to 65535.

Changes in the code:

- **A weekly target.** In `NUMBERS`, before `RETURN` out, add

```
out{"left"} = 40 - out{"total_hours"}
```

and in the template a card next to the hours:

```
<div class="card"><div class="big">{{ left }}</div>hours left</div>
```

- **A second folder to watch**, such as the saved mails of the Inbox Unpacker (M07). In the program, add `"saved": "D:/Mail/saved"` to `sources`; in `NUMBERS`, add

```
DIM saved = FOLDER(sources{"saved"})
out{"saved"} = saved{"files"}
```

and a card with {{ saved }} in the template.

- **Refresh more often.** In the template, the line `<meta http-equiv="refresh" content="300">` reloads the page every 300 seconds. Write 60 for once a minute.

When it goes wrong

- **The browser says it cannot connect:** the program is not running, or it runs on another port. Look for its window or start it again.
- **"Port 8765 is taken",** or the browser shows a page that is not yours: another program uses the port. Set a different port.
- **The numbers do not change:** two copies of the dashboard run, and the browser reaches the older one. Close both windows and start it once.
- **Zero hours although you tracked time:** the dashboard reads the log named in [time_tracker]. Check that the folder there is the one the time tracker writes to.
- **"No page template at":** the templates folder was not copied with the program. Run the wizard again.

Balance dividend

About 20 minutes a week of looking things up, and a Friday view of your week that takes one click.

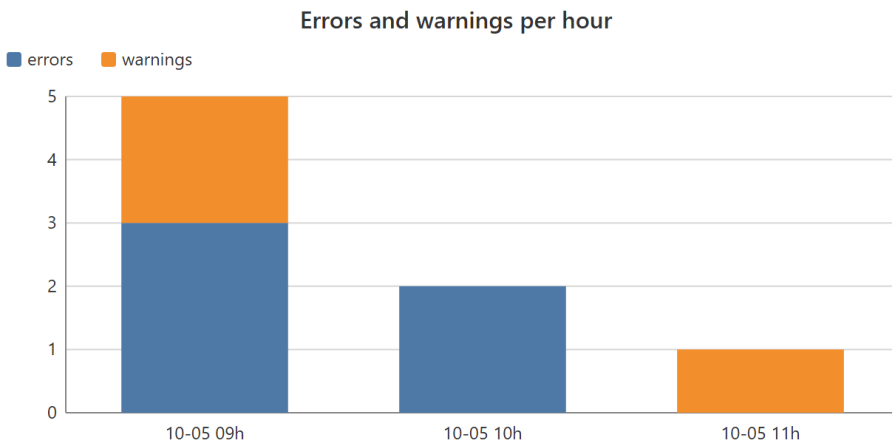
M09 Log Detective

The chore

Jonas's team runs a small import service that brings the orders of the night into the sales system. When something goes wrong, someone opens the log file, scrolls through two thousand lines, and tries to find out whether last night was worse than usual. The answer is always the same search for the word *ERROR*, a few minutes of counting, and a guess. The pattern, such as the same timeout every morning at nine, is only seen by whoever happens to look on three days in a row.

What you get

Every weekday morning the Log Detective reads the log files of a folder and answers three questions. How many errors and warnings were there? In which hour did they pile up? Which error message comes back again and again? It writes a short report and a chart of the hours:



Errors and warnings per hour, drawn by the recipe from the example log

The recipe reads any text log whose lines start with a date and a time, which covers most programs that write logs: web servers, import jobs, backup tools, and the recipes of this book.

Before you start

Find the folder where the program you want to watch writes its logs, and check that you may read it. The wizard asks for that folder and for the pattern of the file names, such as *.log.

The program

The program finds the files, hands each one to the module, and writes the report and the chart:

```
' Log Detective: reads the log files of a folder, counts errors and
' warnings per hour, finds the errors that come again and again, and
' writes a report and a chart. The settings are the [log_detective]
' part of work.conf.
'
'   jdbasic log_detective.jdb           report, text file and chart
'   jdbasic log_detective.jdb --dry-run  the report only

IMPORT WORKCONF, LOGSCAN

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "log_detective."
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", "")
DIM pattern$ = WORKCONF.VALUE(cfg, key$ + "pattern", "*.log")
DIM top_n = WORKCONF.VALUE(cfg, key$ + "top", 5)
DIM work$ = "~/Documents/AutomateWork/logs"
DIM out_dir$ = WORKCONF.FOLDER$(cfg, key$ + "out_folder", work$)
IF folder$ = "" THEN THROW "Set log_detective.folder first."

DIM stats = LOGSCAN.NEW()
DIM files = SORT(DIR$(PATH.JOIN$(folder$, pattern$)))
DIM k = 0
FOR k = 0 TO LEN(files) - 1
    DIM path$ = PATH.JOIN$(folder$, files[k])
    IF NOT FILE.ISDIR(path$) THEN
        LOGSCAN.SCAN_TEXT(stats, TXTREADER$(path$))
    ENDIF
NEXT k

DIM report$ = STR$(LEN(files)) + " files in " + folder$ + CHR$(10)
report$ = report$ + LOGSCAN.REPORT$(stats, top_n)
PRINT report$
IF NOT WORKCONF.FLAG("dry-run") THEN
    MKDIR out_dir$
    DIM text_file$ = PATH.JOIN$(out_dir$, "log_report.txt")
    DIM chart_file$ = PATH.JOIN$(out_dir$, "log_hours.svg")
```

```

    TXTWRITER text_file$, report$ + CHR$(10)
    LOGSCAN.CHART(stats, chart_file$)
    PRINT
    PRINT "Report: "; text_file$
    PRINT "Chart: "; chart_file$
ENDIF

```

The module knows how to read a log line and keeps the counters:

```

' logscan.jdb: application logs read line by line. Errors and warnings
' counted per hour, the same error message found again however its
' numbers change, a short report and a chart of the hours.
'
'   IMPORT LOGSCAN
'   DIM stats = LOGSCAN.NEW()
'   LOGSCAN.SCAN_TEXT(stats, TXTREADER$("app.log"))
'   PRINT LOGSCAN.REPORT$(stats, 5)
'   LOGSCAN.CHART(stats, "hours.svg")

EXPORT MODULE LOGSCAN

IMPORT SVG

DIM gStamp$ = "^([0-9]{4}-[0-9]{2}-[0-9]{2})[ T]([0-9]{2}):[0-9]{2}"
DIM gLevel$ = "\b(FATAL|ERROR|WARNING|WARN)\b"

' Empty counters: "hours" (a map per hour of "ERROR" and "WARN"),
' "messages" (how often each error message came), "lines" and
' "skipped" (lines without a time stamp).
EXPORT FUNC NEW()
    DIM hours AS MAP
    DIM messages AS MAP
    RETURN {"hours": hours, "messages": messages, "lines": 0, _
        "skipped": 0}
ENDFUNC

' A message with every number replaced by #, so "job 4711 failed"
' and "job 4712 failed" count as one.
EXPORT FUNC SIGNATURE$(message$)
    DIM t$ = TRIM$(REGEX.REPLACE("[0-9]+", message$, "#"))
    IF LEN(t$) > 60 THEN t$ = LEFT$(t$, 60)
    RETURN t$
ENDFUNC

' One line as a map with "hour" ("2026-10-05 09"), "level" ("ERROR",
' "WARN" or "") and "message"; an empty map for a line without a time
' stamp at its start.

```

```

EXPORT FUNC PARSE(line$)
    DIM found AS MAP
    DIM stamp = REGEX.FINDALL(gStamp$, line$)
    IF LEN(stamp) = 0 THEN RETURN found
    DIM day$ = stamp[0][0]
    DIM hour$ = stamp[0][1]
    found{"hour"} = day$ + " " + hour$
    found{"level"} = ""
    found{"message"} = ""
    DIM upper$ = UCASE$(line$)
    DIM lv = REGEX.FINDALL(gLevel$, upper$)
    IF LEN(lv) > 0 THEN
        DIM word$ = lv[0][0]
        IF word$ = "FATAL" THEN word$ = "ERROR"
        IF word$ = "WARNING" THEN word$ = "WARN"
        found{"level"} = word$
        DIM at = INSTR(upper$, lv[0][0])
        DIM rest$ = MID$(line$, at + LEN(lv[0][0]), LEN(line$))
        DIM clean$ = REGEX.REPLACE("[^\s-]+", rest$, "")
        found{"message"} = TRIM$(clean$)
    ENDIF
    RETURN found
ENDFUNC

' Adds the lines of a log text to the counters.
EXPORT SUB SCAN_TEXT(stats, text$)
    DIM rows = SPLIT(REPLACE$(text$, CHR$(13), ""), CHR$(10))
    DIM hours = stats{"hours"}
    DIM messages = stats{"messages"}
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM one$ = rows[k]
        IF TRIM$(one$) <> "" THEN
            stats{"lines"} = stats{"lines"} + 1
            DIM p = PARSE(one$)
            IF NOT MAP.EXISTS(p, "hour") THEN
                stats{"skipped"} = stats{"skipped"} + 1
            ELSEIF p{"level"} <> "" THEN
                DIM h$ = p{"hour"}
                IF NOT MAP.EXISTS(hours, h$) THEN
                    hours{h$} = {"ERROR": 0, "WARN": 0}
                ENDIF
                DIM slot = hours{h$}
                DIM lv$ = p{"level"}
                slot{lv$} = slot{lv$} + 1
                IF lv$ = "ERROR" THEN
                    DIM sig$ = SIGNATURE$(p{"message"})
                    IF NOT MAP.EXISTS(messages, sig$) THEN

```

```

        messages{sig$} = 0
    ENDIF
    messages{sig$} = messages{sig$} + 1
ENDIF
ENDIF
ENDIF
NEXT k
ENDSUB

' The total of one level over all hours.
EXPORT FUNC TOTAL(stats, level$)
    DIM hours = stats{"hours"}
    RETURN SUM(SELECT(LAMBDA USE(hours, level$) h$ -> _
        hours{h$}{level$}, MAP.KEYS(hours)))
ENDFUNC

' The error messages, the most frequent first.
EXPORT FUNC TOP_MESSAGES(stats)
    DIM messages = stats{"messages"}
    DIM keys = MAP.KEYS(messages)
    DIM counts = SELECT(LAMBDA USE(messages) m$ -> messages{m$}, keys)
    RETURN SELECT(LAMBDA USE(keys) i -> keys[i], GRADE(-counts))
ENDFUNC

' The busiest hour for errors, "" without any error.
EXPORT FUNC WORST_HOUR$(stats)
    DIM hours = stats{"hours"}
    DIM keys = SORT(MAP.KEYS(hours))
    IF LEN(keys) = 0 THEN RETURN ""
    DIM most = MAX(SELECT(LAMBDA USE(hours) h$ -> _
        hours{h$}{"ERROR"}, keys))
    IF most = 0 THEN RETURN ""
    RETURN FILTER(LAMBDA USE(hours, most) h$ -> _
        hours{h$}{"ERROR"} = most, keys)[0]
ENDFUNC

' The report as text: the totals, the worst hour, the top_n error
' messages and the hours with their counts.
EXPORT FUNC REPORT$(stats, top_n)
    DIM out = []
    DIM errors = TOTAL(stats, "ERROR")
    DIM warnings = TOTAL(stats, "WARN")
    PUSH(out, STR$(stats{"lines"}) + " lines, " + STR$(errors) + _
        " errors, " + STR$(warnings) + " warnings")
    DIM worst$ = WORST_HOUR$(stats)
    IF worst$ <> "" THEN PUSH(out, "Most errors: " + worst$ + ":00")
    PUSH(out, "")
    PUSH(out, "Most frequent errors")

```



```

DIM top = TAKE(top_n, TOP_MESSAGES(stats))
DIM messages = stats{"messages"}
out = APPEND(out, SELECT(LAMBDA USE(messages) s$ -> _
    FORMAT$("{:>5} ", messages{s$}) + s$, top))
PUSH(out, "")
PUSH(out, "Per hour          errors  warnings")
DIM hours = stats{"hours"}
DIM per_hour = SELECT(LAMBDA USE(hours) h$ -> _
    " " + h$ + ":00" + FORMAT$("{:>7}{:>10}", _
    hours{h$}{"ERROR"}, hours{h$}{"WARN"}), SORT(MAP.KEYS(hours)))
out = APPEND(out, per_hour)
RETURN JOIN(out, CHR$(10))
ENDFUNC

' A stacked bar chart of the errors and warnings per hour, as SVG.
EXPORT SUB CHART(stats, path$)
    DIM hours = stats{"hours"}
    DIM keys = SORT(MAP.KEYS(hours))
    DIM labels = SELECT(LAMBDA h$ -> MID$(h$, 5, 8) + "h", keys)
    DIM errors = SELECT(LAMBDA USE(hours) h$ -> hours{h$}{"ERROR"}, _
        keys)
    DIM warnings = SELECT(LAMBDA USE(hours) h$ -> hours{h$}{"WARN"}, _
        keys)
    DIM c = SVG.CHART("stackedbar", 640, 320)
    SVG.SETOPT(c, "title", "Errors and warnings per hour")
    SVG.LABELS(c, labels)
    SVG.SERIES(c, "errors", errors)
    SVG.SERIES(c, "warnings", warnings)
    SVG.WRITEFILE(SVG.RENDER(c), path$)
ENDSUB

```

How it works

1. **The counters.** NEW creates one map that holds everything the recipe learns: a map of hours, each with its count of errors and warnings; a map of error messages with how often each came; and the number of lines read and skipped. Passing this one map around is simpler than passing five separate variables.
1. **Reading a line.** PARSE uses two regular expressions. The first, gStamp\$, looks for a date and a time at the very start of the line and keeps the date and the hour in two groups. REGEX.FINDALL answers a list of matches, each a list of its groups, which is why the code reads stamp[0][0] for the date and stamp[0][1] for the hour. The second expression looks for a level word.

Programs write levels in many ways (WARN, WARNING, [warning], FATAL:), so the line is turned into capitals first, and FATAL and WARNING are mapped to the two levels the report knows.

1. **Lines that belong to the line before.** A stack trace or a long message continues on lines without a time stamp. PARSE answers an empty map for them, and SCAN_TEXT counts them as skipped. The report stays about events, not about lines.
1. **The same error, a different number.** "Timeout after 30 s on job 4711" and "Timeout after 45 s on job 4720" are the same problem. SIGNATURE\$ replaces every number with #, so both become "Timeout after # s on job #" and are counted together. This one line of code is what turns a log into a list of problems.
1. **Putting it together.** TOTAL adds up one level over all hours with SUM over a SELECT. TOP_MESSAGES sorts the error messages by their count with GRADE, as M12 does with meeting hours: the counts are negated so the most frequent comes first. WORST_HOUR\$ takes the MAX of the errors per hour and keeps the first hour with that many with a FILTER. REPORT\$ writes the text, TAKE cuts the list of messages to top, and CHART draws the stacked bar chart with the SVG library: one bar per hour, the errors at the bottom and the warnings on top.

The chart is an SVG file, a drawing any web browser shows. Double-click it, or put it on a page next to the report.

Run it

With the example log of the recipe's folder:

```
jdbasic log_detective.jdb
1 files in C:/Logs/import
11 lines, 5 errors, 3 warnings
Most errors: 2026-10-05 09:00

Most frequent errors
  3 Timeout after # s on job #
  1 Cannot open file report_#.xlsx
  1 database connection lost

Per hour          errors  warnings
```

2026-10-05 09:00	3	2
2026-10-05 10:00	2	0
2026-10-05 11:00	0	1

The last two lines name the text file and the chart. With `--dry-run` the recipe prints the report and writes nothing.

Schedule it

The wizard plans the Log Detective for weekdays at 7:45, so the report of the night is waiting when the team starts.

Make it yours

The settings are in the part of `work.conf` that starts with `[log_detective]`:

```
[log_detective]
folder = "C:/Logs/import"
pattern = "*.log"
top = 5
out_folder = "~/Documents/AutomateWork/logs"
```

The interesting changes are in the code:

- **A different time stamp.** Some programs write `05.10.2026 09:14:03` or `[05/Oct/2026:09:14:03]`. Change `gStamp$` at the top of the module. For the German form it becomes:

```
DIM gStamp$ = "^([0-9]{2}\.[0-9]{2}\.[0-9]{4}) ([0-9]{2}):"
```

The two groups still hold the day and the hour, so nothing else has to change.

- **More levels.** To count `CRITICAL` as an error, add it to `gLevel$` (`"\b(FATAL|CRITICAL|ERROR|WARNING|WARN)\b"`) and map it in `PARSE` with one more line: `IF word$ = "CRITICAL" THEN word$ = "ERROR"`.
- **Ignore a known message.** If a harmless error fills the report, skip it in `SCAN_TEXT` before it is counted:

```
IF INSTR(p{"message"}, "heartbeat") >= 0 THEN CONTINUEFOR
```

- **Only the last day.** Logs that grow for weeks make the report long. Compare `p{"hour"}` with yesterday's date (from `FORMAT_DATE`) in `SCAN_TEXT` and count only matching lines.

When it goes wrong

- **"0 errors" although you see errors:** the time stamp has a form `gStamp$` does not know, so every line is skipped. The report shows the number of lines; compare it with what you expect, and adapt the expression as shown above.
- **The file is in use:** some programs keep their log open. Reading it usually works anyway. If not, point the recipe at the rotated files of the days before (`app.log.1`, `app-2026-10-04.log`) with the pattern.
- **Very large logs take long:** the recipe reads each file whole. For logs of many megabytes, schedule it at night and keep only the files of the last days in the folder.

Balance dividend

About 30 minutes a week of scrolling through logs, and problems seen on the first morning instead of the third.

M10 Hot Folder

The chore

Three times a day a colleague from sales drops a file into Jonas's shared folder: an order list as CSV, a scanned delivery note, a price list. Each one needs the same treatment. The CSV file has to become a workbook, since that is what the team opens. Every file gets a name with the date in front, so the folder sorts by day. Then it moves to the archive. None of this is hard, but it waits until someone looks at the folder, and on a busy day nobody does.

What you get

A folder that works by itself. Every five minutes the Hot Folder looks into it. Each file that has finished arriving is handled by the first rule that fits it:

- a CSV file becomes a workbook with a frozen header row, and the original CSV file is kept in a subfolder `originals`;
- every other file is moved on;
- both get a new name such as `2026-10-05 orders.xlsx`.

Each step is written to a log, so a question like *where did the price list of Tuesday go?* has an answer. Your own rules can send PDF files to one folder and price lists to another.

Before you start

Pick or create the folder people drop files into, and a folder for the handled files. The done folder may be inside the watched folder; the recipe leaves folders alone. Make sure the account that runs the scheduled task may write to both.

The program

The program reads the settings and takes the lock. Then it lets the module plan and carry out the work:

```

' Hot Folder: one look at a folder. Every file that has settled is
' handled by the first rule that fits it, and each step lands in a
' log. Scheduled every few minutes, with a lock file so two runs never
' overlap. The settings are the [hot_folder] part of work.conf.
,
'   jdbasic hot_folder.jdb           handle the files
'   jdbasic hot_folder.jdb --dry-run  show what it would do

IMPORT WORKCONF, HOTFOLDER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "hot_folder."
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", "")
DIM done$ = WORKCONF.FOLDER$(cfg, key$ + "done_folder", "")
DIM rename$ = WORKCONF.VALUE(cfg, key$ + "rename", "{date} {name}")
DIM settle = WORKCONF.VALUE(cfg, key$ + "settle_seconds", 60)
DIM sep$ = WORKCONF.VALUE(cfg, key$ + "separator", ";")
IF folder$ = "" THEN THROW "Set hot_folder.folder first."
IF done$ = "" THEN done$ = PATH.JOIN$(folder$, "done")
DIM rules = HOTFOLDER.DEFAULT_RULES(done$, rename$)
DIM own = WORKCONF.VALUE(cfg, key$ + "rule", [])
IF LEN(own) > 0 THEN rules = own

DIM lock$ = PATH.JOIN$(folder$, ".hot_folder.lock")
DIM log$ = PATH.JOIN$(done$, "hot_folder.log")
IF WORKCONF.FLAG("dry-run") THEN
    DIM jobs = HOTFOLDER.PLAN(folder$, rules, settle, NOW())
    DIM k = 0
    FOR k = 0 TO LEN(jobs) - 1
        DIM job = jobs[k]
        PRINT "would handle "; PATH.BASENAME$(job{"from"}); " -> "; _
            job{"to"}
    NEXT k
    IF LEN(jobs) = 0 THEN PRINT "Nothing to do."
ELSEIF HOTFOLDER.LOCK(lock$, NOW(), 30) THEN
    TRY
        DIM todo = HOTFOLDER.PLAN(folder$, rules, settle, NOW())
        DIM lines = HOTFOLDER.APPLY(todo, sep$, log$, NOW())
        IF LEN(lines) = 0 THEN PRINT "Nothing to do."
        IF LEN(lines) > 0 THEN PRINT JOIN(lines, CHR$(10))
    CATCH
        PRINT "Stopped: "; ERRMSG$
    ENDTRY
    HOTFOLDER.RELEASE(lock$)
ELSE
    PRINT "Another run is working on the folder."
ENDIF

```

The module holds the rules, the lock and the work on each file:

```
' hotfolder.jdb: a folder that works by itself. Files that land in it
' are matched against rules, renamed, converted from CSV to a workbook
' and moved on, with a line in a log for each. A lock file keeps two
' runs from working on the folder at the same time.
'
'   IMPORT HOTFOLDER
'   DIM rules = HOTFOLDER.DEFAULT_RULES("C:/Done", "{date} {name}")
'   IF HOTFOLDER.LOCK(lock$, NOW(), 30) THEN
'       DIM jobs = HOTFOLDER.PLAN("C:/Inbox", rules, 60, NOW())
'       HOTFOLDER.APPLY(jobs, ";", "C:/Done/hot_folder.log", NOW())
'       HOTFOLDER.RELEASE(lock$)
'   ENDIF

EXPORT MODULE HOTFOLDER

IMPORT XLSX

' The regular expression for a file pattern with * and ?.
FUNC GlobRegex$(pattern$)
    DIM chars = -LCASE$(pattern$)
    DIM out$ = "^"
    DIM k = 0
    FOR k = 0 TO LEN(chars) - 1
        DIM ch$ = chars[k]
        IF ch$ = "*" THEN
            out$ = out$ + ".*"
        ELSEIF ch$ = "?" THEN
            out$ = out$ + "."
        ELSEIF LEN(REGEX.FINDALL("[a-z0-9 _]", ch$)) > 0 THEN
            out$ = out$ + ch$
        ELSE
            out$ = out$ + "[" + ch$ + "]"
        ENDIF
    NEXT k
    RETURN out$ + "$"
ENDFUNC

' TRUE when a file name fits a pattern such as "*.csv" or
' "invoice_?.pdf", in small or capital letters.
EXPORT FUNC MATCHES(file_name$, pattern$)
    DIM rx$ = GlobRegex$(pattern$)
    RETURN LEN(REGEX.FINDALL(rx$, LCASE$(file_name$))) > 0
ENDFUNC

' Two rules: CSV files become workbooks, every other file is moved,
' both into target$ and renamed by rename$.
```

```

EXPORT FUNC DEFAULT_RULES(target$, rename$)
  DIM csv = {"match": "*.csv", "convert": "xlsx", _
    "rename": rename$, "target": target$}
  DIM rest = {"match": "*", "convert": "", "rename": rename$, _
    "target": target$}
  RETURN [csv, rest]
ENDFUNC

' The first rule whose "match" fits the file name, an empty map when
' no rule does.
EXPORT FUNC RULE_FOR(rules, file_name$)
  DIM k = 0
  FOR k = 0 TO LEN(rules) - 1
    DIM r = rules[k]
    IF MATCHES(file_name$, r{"match"}) THEN RETURN r
  NEXT k
  DIM none AS MAP
  RETURN none
ENDFUNC

' A new file name from a pattern and an extension: {name} is the old
' name without its extension, {date} and {time} the day and time the
' file last changed, from a time such as "2026-10-05 09:14:03".
EXPORT FUNC NEW_NAME$(pattern$, file_name$, changed$, ext$)
  DIM old_ext$ = PATH.EXT$(file_name$)
  DIM stem$ = LEFT$(file_name$, LEN(file_name$) - LEN(old_ext$))
  IF pattern$ = "" THEN RETURN stem$ + ext$
  DIM clock$ = REPLACE$(MID$(changed$, 11, 5), ":", "-")
  DIM s$ = REPLACE$(pattern$, "{name}", stem$)
  s$ = REPLACE$(s$, "{date}", LEFT$(changed$, 10))
  s$ = REPLACE$(s$, "{time}", clock$)
  RETURN s$ + ext$
ENDFUNC

' TRUE when this run may work on the folder: there is no lock file,
' or one older than stale_minutes that a stopped run left behind.
' Writes the lock with the time now.
EXPORT FUNC LOCK(path$, now, stale_minutes)
  IF FILE.EXISTS(path$) THEN
    DIM since$ = TRIM$(TXTREADER$(path$))
    DIM age = DATEDIFF("N", CDATE(since$), now)
    IF age < stale_minutes THEN RETURN FALSE
  ENDIF
  TXTWRITER path$, FORMAT_DATE(now, "%Y-%m-%d %H:%M:%S")
  RETURN TRUE
ENDFUNC

' Removes the lock file.

```



```

EXPORT SUB RELEASE(path$)
  IF FILE.EXISTS(path$) THEN KILL path$
ENDSUB

' The jobs for the files in a folder: a map per file with "from", "to"
' and "convert". Files changed less than settle_seconds ago are still
' being copied and wait for the next run, as do files no rule matches
' and names that start with a dot.
EXPORT FUNC PLAN(folder$, rules, settle_seconds, now)
  DIM jobs = []
  DIM entries = DIR$(PATH.JOIN$(folder$, "*"))
  DIM k = 0
  FOR k = 0 TO LEN(entries) - 1
    DIM file_name$ = entries[k]
    DIM from$ = PATH.JOIN$(folder$, file_name$)
    DIM info = FILE.STAT(from$)
    DIM changed$ = info{"mtime"}
    DIM age = DATEDIFF("S", CDATE(changed$), now)
    DIM r = RULE_FOR(rules, file_name$)
    DIM usable = LEFT$(file_name$, 1) <> "." ANDALSO _
      NOT info{"is_dir"} ANDALSO MAP.EXISTS(r, "match")
    IF usable ANDALSO age >= settle_seconds THEN
      DIM ext$ = PATH.EXT$(file_name$)
      IF r{"convert"} = "xlsx" THEN ext$ = ".xlsx"
      DIM new$ = NEW_NAME$(r{"rename"}, file_name$, changed$, _
        ext$)
      PUSH(jobs, {"from": from$, "convert": r{"convert"}, _
        "to": PATH.JOIN$(r{"target"}, new$)})
    ENDIF
  NEXT k
  RETURN jobs
ENDFUNC

' The fields of one CSV line, with quoted fields that hold the
' separator or doubled quotes. Fields that are plain numbers become
' numbers.
EXPORT FUNC CSV_FIELDS(line$, sep$)
  DIM out = []
  DIM cur$ = ""
  DIM quoted = FALSE
  DIM q$ = CHR$(34)
  DIM chars = -line$
  DIM k = 0
  FOR k = 0 TO LEN(chars)
    DIM ch$ = sep$
    IF k < LEN(chars) THEN ch$ = chars[k]
    DIM after$ = ""
    IF k + 1 < LEN(chars) THEN after$ = chars[k + 1]

```

```

    IF quoted AND ALSO ch$ = q$ AND ALSO after$ = q$ THEN
        cur$ = cur$ + q$
        k = k + 1
    ELSEIF quoted AND ALSO ch$ = q$ THEN
        quoted = FALSE
    ELSEIF quoted THEN
        cur$ = cur$ + ch$
    ELSEIF ch$ = q$ THEN
        quoted = TRUE
    ELSEIF ch$ = sep$ THEN
        DIM plain = REGEX.FINDALL("^-?[0-9]+([.][0-9]+)?$", cur$)
        IF LEN(plain) > 0 THEN
            PUSH(out, VAL(cur$))
        ELSE
            PUSH(out, cur$)
        ENDIF
        cur$ = ""
    ELSE
        cur$ = cur$ + ch$
    ENDIF
NEXT k
RETURN out
ENDFUNC

' A CSV file written as a workbook with one sheet "Data" and a frozen
' header row. Answers the number of rows below the header.
EXPORT FUNC CSV_TO_XLSX(src$, dest$, sep$)
    DIM text$ = REPLACE$(TXTREADER$(src$), CHR$(13), "")
    DIM lines = SPLIT(text$, CHR$(10))
    DIM filled = FILTER(LAMBDA l$ -> TRIM$(l$) <> "", lines)
    DIM rows = SELECT(LAMBDA USE(sep$) l$ -> CSV_FIELDS(l$, sep$), _
        filled)
    DIM wb = XLSX.NEW()
    DIM sheet = XLSX.SHEET(wb, "Data", rows)
    XLSX.FREEZE(sheet, 1)
    XLSX.WRITE(dest$, wb)
    RETURN LEN(rows) - 1
ENDFUNC

' A path that is free: the path itself, or with " (2)", " (3)" and so
' on before the extension.
FUNC FreePath$(path$)
    IF NOT FILE.EXISTS(path$) THEN RETURN path$
    DIM ext$ = PATH.EXT$(path$)
    DIM stem$ = LEFT$(path$, LEN(path$) - LEN(ext$))
    DIM n = 2
    DO WHILE FILE.EXISTS(stem$ + " (" + STR$(n) + ")" + ext$)
        n = n + 1

```

```

    LOOP
    RETURN stem$ + " (" + STR$(n) + ")" + ext$
ENDFUNC

' Carries out the jobs and adds a line per file to the log. A CSV
' file that becomes a workbook is kept in the subfolder "originals"
' of its target. A job that fails is logged and the rest go on.
' Answers the log lines.
EXPORT FUNC APPLY(jobs, sep$, log$, now)
    DIM stamp$ = FORMAT_DATE(now, "%Y-%m-%d %H:%M")
    DIM lines = []
    DIM k = 0
    FOR k = 0 TO LEN(jobs) - 1
        DIM job = jobs[k]
        DIM from$ = job{"from"}
        DIM to$ = FreePath$(job{"to"})
        DIM new$ = PATH.BASENAME$(to$)
        DIM what$ = ""
        TRY
            MKDIR PATH.DIRNAME$(to$)
            IF job{"convert"} = "xlsx" THEN
                DIM n = CSV_TO_XLSX(from$, to$, sep$)
                DIM there$ = PATH.DIRNAME$(to$)
                DIM keep$ = PATH.JOIN$(there$, "originals")
                MKDIR keep$
                DIM old$ = PATH.JOIN$(keep$, PATH.BASENAME$(from$))
                FILE.MOVE(from$, FreePath$(old$))
                what$ = "made " + new$ + " (" + STR$(n) + " rows)"
            ELSE
                FILE.MOVE(from$, to$)
                what$ = "moved to " + new$
            ENDIF
        CATCH
            what$ = "failed: " + ERRMSG$
        ENDTRY
        DIM src$ = PATH.BASENAME$(from$)
        PUSH(lines, stamp$ + " " + src$ + " " + what$)
    NEXT k
    IF LEN(lines) > 0 THEN
        DIM old_log$ = ""
        IF FILE.EXISTS(log$) THEN old_log$ = TXTREADER$(log$)
        MKDIR PATH.DIRNAME$(log$)
        TXTWRITER log$, old_log$ + JOIN(lines, CHR$(10)) + CHR$(10)
    ENDIF
    RETURN lines
ENDFUNC

```

How it works

1. **One look, many times.** The recipe does not run all day and watch the folder. It looks once, does what there is to do, and ends. The task scheduler starts it again five minutes later. A program that ends after each look cannot hang for days unnoticed, holds no memory, and survives a restart of the computer without anyone remembering to start it again.
1. **The lock file.** A run may take longer than five minutes, for a large CSV file on a slow network drive. The next run must not start on the same files then. LOCK writes the file `.hot_folder.lock` with the current time into the watched folder and answers TRUE. A second run finds the file and answers FALSE, and the program prints *Another run is working on the folder* and ends. RELEASE removes the file when the work is done. If a run is stopped halfway, by a crash or a shutdown, its lock stays behind. That is why the lock holds the time: a lock older than 30 minutes is taken over.
1. **Files that are still arriving.** A large file is copied into the folder over several seconds. A run that picks it up halfway would convert half a file. PLAN reads the time each file was last changed with `FILE.STAT` and leaves every file alone that changed less than `settle_seconds` ago. The next run handles it.
1. **Rules with patterns.** A rule is a map with four keys: `match`, a file pattern such as `*.csv`; `convert`, either `xlsx` or empty; `rename`, a name pattern; and `target`, the folder the file goes to. MATCHES turns the file pattern into a regular expression: `*` becomes `.`, `?` becomes `.`, and every other character that is no letter or digit is put in brackets, so a dot in `*.csv` stands for a dot and not for any character. RULE_FOR answers the first rule that fits, which is why the catch-all rule `*` comes last.
1. **The CSV conversion.** CSV_FIELDS reads a line character by character, so a quoted field like "Weber; Sohn & Co" keeps its semicolon, and two quotes in a row stand for one. A field made of digits with an optional decimal point becomes a number, which lets Excel add up the column. CSV_TO_XLSX drops the empty lines with a FILTER, splits the others with a SELECT over CSV_FIELDS and writes the rows as a sheet.
1. **Never lose a file.** FreePath\$ adds (2) to a name that is already taken in the target folder, so a second `orders.csv` on the same day does not overwrite the first. APPLY wraps the work on each file in TRY and CATCH: a file that

cannot be moved, maybe because it is open in Excel, gets a *failed* line in the log and stays where it is, and the other files are handled.

Run it

With the example CSV file and a scanned PDF in the folder:

```
jdbasic hot_folder.jdb --dry-run
would handle orders.csv -> C:/Orders/done/2026-10-05 orders.xlsx
would handle scan.pdf -> C:/Orders/done/2026-10-05 scan.pdf
```

Without `--dry-run`, each line of the log is printed too:

```
jdbasic hot_folder.jdb
2026-10-05 09:20 orders.csv made 2026-10-05 orders.xlsx (3 rows)
2026-10-05 09:20 scan.pdf moved to 2026-10-05 scan.pdf
```

A run that finds nothing prints *Nothing to do*. That is the common case, and it costs a fraction of a second.

Schedule it

The wizard plans the Hot Folder every 5 minutes. Shorter intervals make little sense for files people drop by hand. For a folder that another program fills at night, once an hour is enough.

Make it yours

The settings are in the part of `work.conf` that starts with `[hot_folder]`:

```
[hot_folder]
folder = "C:/Orders"
done_folder = "C:/Orders/done"
rename = "{date} {name}"
separator = ";"
settle_seconds = 60
```

`rename` may use `{date}` and `{time}` of the last change and `{name}`, the old name without its extension. An empty `rename` keeps the names.

- **Your own rules.** Rules in `work.conf` replace the two built-in ones. Each rule needs all four keys, and the first rule that fits a file wins:

```
[[hot_folder.rule]]
match = "*.pdf"
convert = ""
rename = "{name}"
target = "C:/Scans"

[[hot_folder.rule]]
match = "*.csv"
convert = "xlsx"
rename = "{date} {name}"
target = "C:/Orders/done"
```

Files that no rule fits stay in the folder.

- **A month in the name.** Add one line to `NEW_NAME$`, next to the other placeholders, and use `{month}` in `rename`:

```
s$ = REPLACE$(s$, "{month}", LEFT$(changed$, 7))
```

- **A folder per month.** To sort the handled files into subfolders such as 2026-10, change the target path in `PLAN`:

```
DIM month$ = LEFT$(changed$, 7)
DIM there$ = PATH.JOIN$(r{"target"}, month$, new$)
```

and use `there$` as the "to" of the job. `APPLY` creates missing folders on the way.

When it goes wrong

- **"Another run is working on the folder", every time:** a lock is left over and the stale time has not passed yet. Wait 30 minutes or delete `.hot_folder.lock` in the watched folder.
- **A file stays in the folder:** no rule fits it, or it is still changing. Run with `--dry-run`, which lists every file the recipe would handle.
- **A failed line in the log:** the file was open in another program, or the target folder is not writable. The file stays and is tried again on the next run.
- **Every row is one long field in the workbook:** the CSV file uses a different separator. Set separator in `work.conf`.

Balance dividend

About 20 minutes a week of renaming, converting and moving, and files that are ready minutes after they arrive instead of whenever someone looks.

M11 Data Cleaner

The chore

Every Monday Jonas gets the order list of the week as an export from the shop system, and every Monday it needs half an hour of repair before anyone can work with it. The dates come in three forms, because some orders were typed in by hand. Amounts arrive as text with a decimal comma, so Excel will not add them up. The same customer is written as *Müller GmbH* in one row and *Mueller GmbH* in the next. A few rows appear twice, and every few weeks an IBAN has a typo that nobody sees until the bank sends the payment back.

What you get

The Data Cleaner reads the export, a CSV file or a workbook, and repairs what it can:

- dates in any of the usual forms become ISO dates (2026-10-05);
- amounts with a decimal comma or point, thousands separators, a currency sign or a minus behind become real numbers;
- spaces are tidied, IBANs are written in groups of four, mail addresses in small letters;
- rows that appear twice are dropped.

What it cannot repair it lists: a date that does not exist, an IBAN whose check digits are wrong, a mail address without a domain. It also lists customer names that look like the same customer written twice. Everything lands in one workbook with three sheets: *Clean*, the repaired table, *Problems*, one row per cell to look at, and *Similar names*.

Before you start

Save one export in your work folder and open it in a text editor. Two things matter. The first is the separator between the fields: a semicolon in most European exports, a comma in American ones. The second is the order of day and month in dates such as 05/10/2026. The wizard asks for both, and for the column that holds the customer names.

The program

The program reads the settings, decides the kind of each column, and prints what the module found:

```
' Data Cleaner: a messy export (CSV or workbook) cleaned column by
' column, repeated rows dropped, names that look alike found, and a
' clean workbook written with the problems on a sheet of their own.
' The settings are the [data_cleaner] part of work.conf.
'
'   jdbasic data_cleaner.jdb                clean and write the workbook
'   jdbasic data_cleaner.jdb --dry-run      show what it would change

IMPORT WORKCONF, CLEANER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "data_cleaner."
DIM input$ = WORKCONF.FOLDER$(cfg, key$ + "input", "")
DIM output$ = WORKCONF.FOLDER$(cfg, key$ + "output", "")
DIM sep$ = WORKCONF.VALUE(cfg, key$ + "separator", ";")
DIM name_col$ = WORKCONF.VALUE(cfg, key$ + "name_column", "")
DIM cutoff = WORKCONF.VALUE(cfg, key$ + "similar", 85)
DIM opts AS MAP
opts{"day_first"} = WORKCONF.VALUE(cfg, key$ + "day_first", TRUE)
opts{"decimal"} = WORKCONF.VALUE(cfg, key$ + "decimal", "auto")
IF input$ = "" THEN THROW "Set data_cleaner.input first."
IF output$ = "" THEN
    DIM ext$ = PATH.EXT$(input$)
    output$ = LEFT$(input$, LEN(input$) - LEN(ext$)) + "_clean.xlsx"
ENDIF

DIM rows = CLEANER.READ_TABLE(input$, sep$)
DIM spec = CLEANER.GUESS_KINDS(rows[0])
DIM given = WORKCONF.VALUE(cfg, key$ + "columns", spec)
DIM cols = MAP.KEYS(given)
DIM k = 0
FOR k = 0 TO LEN(cols) - 1
    DIM col$ = cols[k]
    spec{col$} = given{col$}
NEXT k
DIM result = CLEANER.CLEAN(rows, spec, opts)
DIM similar = []
IF name_col$ <> "" THEN
    similar = CLEANER.SIMILAR(result{"rows"}, name_col$, cutoff)
ENDIF

PRINT "Rows read: "; LEN(rows) - 1
PRINT "Cells changed: "; result{"changed"}
```

```

PRINT "Repeated rows dropped: "; result{"duplicates"}
DIM problems = result{"problems"}
PRINT "Problems: "; LEN(problems)
FOR k = 0 TO LEN(problems) - 1
    DIM p = problems[k]
    PRINT "  row "; p[0]; ", "; p[1]; ": "; p[3]; " ("; p[2]; ")"
NEXT k
PRINT "Names that look alike: "; LEN(similar)
FOR k = 0 TO LEN(similar) - 1
    DIM s = similar[k]
    PRINT "  "; s[0]; " / "; s[1]; " ("; s[2]; ")"
NEXT k
IF NOT WORKCONF.FLAG("dry-run") THEN
    CLEANER.WRITE_XLSX(result, similar, output$)
    PRINT "Clean workbook: "; output$
ENDIF

```

The module does the cleaning, one cell at a time, and writes the workbook:

```

' cleaner.jdb: messy exports made usable. Dates in several forms to
' ISO, numbers with decimal comma or point, spaces tidied, IBANs and
' mail addresses checked, repeated rows dropped, and names that look
' like the same customer written twice found.
'
'   IMPORT CLEANER
'   DIM spec = {"Date": "date", "Amount": "number"}
'   DIM result = CLEANER.CLEAN(rows, spec, {"day_first": TRUE})
'   CLEANER.WRITE_XLSX(result, [], "clean.xlsx")

EXPORT MODULE CLEANER

IMPORT DT, FUZZY, VALID, XLSX

DIM gMonths = ["jan", "feb", "mar", "apr", "may", "jun", "jul", _
    "aug", "sep", "oct", "nov", "dec"]

DIM gIso$ = "^([0-9]{4})[-./]([0-9]{1,2})[-./]([0-9]{1,2})"
DIM gDmy$ = "^([0-9]{1,2})([./-])([0-9]{1,2})[./-]([0-9]{2,4})$"
DIM gDayName$ = "^([0-9]{1,2})\\.? ([a-z]{3})[a-z]*\\.? ([0-9]{4})$"
DIM gNameDay$ = "^([a-z]{3})[a-z]*\\.? ([0-9]{1,2}),? ([0-9]{4})$"

' 1 to 12 for the first three letters of a month name, 0 for others.
FUNC MonthNo(s$)
    DIM k = 0
    FOR k = 0 TO 11
        IF gMonths[k] = s$ THEN RETURN k + 1
    NEXT k

```

```

    RETURN 0
ENDFUNC

FUNC Two$(n)
    RETURN RIGHT$("0" + STR$(n), 2)
ENDFUNC

' "YYYY-MM-DD" for parts that make a real date, else "".
FUNC Ymd$(y, m, d)
    DIM year = y
    IF year < 100 THEN year = year + 2000
    IF m < 1 ORELSE m > 12 ORELSE d < 1 THEN RETURN ""
    IF d > DT.DAYS_IN(year, m) THEN RETURN ""
    RETURN STR$(year) + "-" + Two$(m) + "-" + Two$(d)
ENDFUNC

' A date as YYYY-MM-DD from any of the forms exports use: an Excel
' serial number, 2026-10-05, 05.10.2026, 5/10/26, 5 Oct 2026 or
' Oct 5, 2026. day_first reads 05/10 as the 5th of October. Answers
' "" for a text that is no date.
EXPORT FUNC ISO_DATE$(raw, day_first)
    IF TYPEOF(raw) <> "STRING" THEN
        DIM serial = FLOOR(raw)
        IF serial < 1 ORELSE serial > 2958465 THEN RETURN ""
        DIM t = DT.MAKE(1899, 12, 30) + serial * 86400
        RETURN DT.FMT$(t, "%Y-%m-%d", 0)
    ENDIF
    DIM s$ = LCASE$(TRIM$(raw))
    DIM g = REGEX.FINDALL(gIso$, s$)
    IF LEN(g) > 0 THEN
        RETURN Ymd$(VAL(g[0][0]), VAL(g[0][1]), VAL(g[0][2]))
    ENDIF
    g = REGEX.FINDALL(gDmy$, s$)
    IF LEN(g) > 0 THEN
        DIM a = VAL(g[0][0])
        DIM b = VAL(g[0][2])
        DIM y = VAL(g[0][3])
        IF g[0][1] = "." ORELSE day_first THEN RETURN Ymd$(y, b, a)
        RETURN Ymd$(y, a, b)
    ENDIF
    g = REGEX.FINDALL(gDayName$, s$)
    IF LEN(g) > 0 THEN
        DIM mo = MonthNo(g[0][1])
        RETURN Ymd$(VAL(g[0][2]), mo, VAL(g[0][0]))
    ENDIF
    g = REGEX.FINDALL(gNameDay$, s$)
    IF LEN(g) > 0 THEN
        DIM mn = MonthNo(g[0][0])

```

```

        RETURN Ymd$(VAL(g[0][2]), mn, VAL(g[0][1]))
    ENDIF
    RETURN ""
ENDFUNC

' A number from a text with a decimal comma or point, thousands
' separators, a currency, a sign in front, a minus behind (as SAP
' writes it) or brackets around. decimal$ is ",", "." or "auto", which
' takes the last separator as the decimal one, and a lone one too.
' Answers NONE for a text that is no number.
EXPORT FUNC NUMBER(raw, decimal$)
    IF TYPEOF(raw) <> "STRING" THEN RETURN raw
    DIM s$ = REPLACE$(TRIM$(raw), CHR$(194) + CHR$(160), "")
    s$ = REGEX.REPLACE("\s|€|\$|£|EUR|USD|CHF", s$, "")
    DIM minus = FALSE
    IF STARTSWITH(s$, "(") ANDALSO ENDSWITH(s$, ")") THEN
        minus = TRUE
        s$ = MID$(s$, 1, LEN(s$) - 2)
    ENDIF
    IF ENDSWITH(s$, "-") THEN
        minus = TRUE
        s$ = LEFT$(s$, LEN(s$) - 1)
    ENDIF
    DIM shape = REGEX.FINDALL("^[+-]?[0-9][0-9.]*$", s$)
    IF LEN(shape) = 0 THEN RETURN NONE
    DIM dec$ = decimal$
    IF dec$ = "auto" THEN
        dec$ = "."
        DIM comma = CINT(INSTR(s$, ","))
        DIM point = CINT(INSTR(s$, "."))
        DIM last_comma = comma >= 0 ANDALSO comma > point
        IF last_comma THEN dec$ = ","
    ENDIF
    DIM thousands$ = ","
    IF dec$ = "," THEN thousands$ = "."
    s$ = REPLACE$(REPLACE$(s$, thousands$, ""), dec$, ".")
    shape = REGEX.FINDALL("^[+-]?[0-9]+(\\.[0-9]+)?$", s$)
    IF LEN(shape) = 0 THEN RETURN NONE
    DIM v = VAL(s$)
    IF minus THEN v = -v
    RETURN v
ENDFUNC

' A text with its spaces tidied: none at the ends, one between words.
EXPORT FUNC TIDY$(raw)
    DIM s$ = "" + raw
    RETURN TRIM$(REGEX.REPLACE("\s+", s$, " "))
ENDFUNC

```

```

EXPORT FUNC IS_MAIL(s$)
    DIM p$ = "^[@\s]+@[^\s]+\.[a-zA-Z]{2,}$"
    RETURN LEN(REGEX.FINDALL(p$, s$)) > 0
ENDFUNC

FUNC Answer(v, why$)
    RETURN {"value": v, "problem": why$}
ENDFUNC

' One cell cleaned for its kind: a map with "value" and "problem", ""
' when the cell is fine. Empty cells stay empty, and the kind "keep"
' leaves the cell as it is.
EXPORT FUNC CELL(raw, kind$, opts)
    IF kind$ = "keep" THEN RETURN Answer(raw, "")
    IF TYPEOF(raw) = "NONE" THEN RETURN Answer("", "")
    IF TIDY$(raw) = "" THEN RETURN Answer("", "")
    IF kind$ = "date" THEN
        DIM d$ = ISO_DATE$(raw, opts{"day_first"})
        IF d$ = "" THEN RETURN Answer(raw, "not a date")
        RETURN Answer(d$, "")
    ENDIF
    IF kind$ = "number" THEN
        DIM v = NUMBER(raw, opts{"decimal"})
        IF TYPEOF(v) = "NONE" THEN RETURN Answer(raw, "not a number")
        RETURN Answer(v, "")
    ENDIF
    DIM t$ = TIDY$(raw)
    IF kind$ = "iban" THEN
        IF NOT VALID.ISIBAN(t$) THEN
            RETURN Answer(t$, "not a valid IBAN")
        ENDIF
        RETURN Answer(VALID.IBAN$(t$), "")
    ENDIF
    IF kind$ = "mail" THEN
        IF NOT IS_MAIL(t$) THEN
            RETURN Answer(t$, "not a mail address")
        ENDIF
        RETURN Answer(LCASE$(t$), "")
    ENDIF
    RETURN Answer(t$, "")
ENDFUNC

' A kind for each column guessed from its name: "date" when the name
' holds dat (Date, Datum), "number" for amount, betrag, price, preis,
' sum or total, "iban", "mail", and "text" for the rest.
EXPORT FUNC GUESS_KINDS(header)
    DIM spec AS MAP

```

```

DIM c = 0
FOR c = 0 TO LEN(header) - 1
    DIM col$ = "" + header[c]
    DIM low$ = LCASE$(col$)
    DIM kind$ = "text"
    IF INSTR(low$, "dat") >= 0 THEN kind$ = "date"
    DIM money$ = "amount|betrag|price|preis|sum|total"
    IF LEN(REGEX.FINDALL(money$, low$)) > 0 THEN kind$ = "number"
    IF INSTR(low$, "iban") >= 0 THEN kind$ = "iban"
    IF INSTR(low$, "mail") >= 0 THEN kind$ = "mail"
    spec{col$} = kind$
NEXT c
RETURN spec
ENDFUNC

' A table cleaned: rows[0] holds the column names, spec maps a name to
' its kind (date, number, text, iban, mail; others stay as they are).
' Answers "rows" (the clean table), "problems" (rows of row, column,
' value, reason), "changed" (cells that changed) and "duplicates"
' (rows dropped as equal to a row before them).
EXPORT FUNC CLEAN(rows, spec, opts)
    DIM header = rows[0]
    DIM out = [header]
    DIM problems = []
    DIM seen AS MAP
    DIM changed = 0
    DIM dups = 0
    DIM r = 0
    DIM c = 0
    FOR r = 1 TO LEN(rows) - 1
        DIM src = rows[r]
        DO WHILE LEN(src) < LEN(header)
            PUSH(src, "")
        LOOP
        DIM clean_row = []
        DIM key$ = ""
        FOR c = 0 TO LEN(header) - 1
            DIM col$ = header[c]
            DIM kind$ = "keep"
            IF MAP.EXISTS(spec, col$) THEN kind$ = spec{col$}
            DIM res = CELL(src[c], kind$, opts)
            DIM was$ = "" + src[c]
            DIM now$ = "" + res{"value"}
            IF res{"problem"} <> "" THEN
                PUSH(problems, [r + 1, col$, was$, res{"problem"}])
            ENDIF
            IF now$ <> was$ THEN changed = changed + 1
            PUSH(clean_row, res{"value"})
        
```

```

        key$ = key$ + LCASE$(now$) + "|"
    NEXT c
    IF MAP.EXISTS(seen, key$) THEN
        dups = dups + 1
    ELSE
        seen{key$} = TRUE
        PUSH(out, clean_row)
    ENDIF
NEXT r
RETURN {"rows": out, "problems": problems, "changed": changed, _
        "duplicates": dups}
ENDFUNC

' Pairs of different names in a column that score at least cutoff
' (0 to 100) as the same text: rows of [name, other, score].
EXPORT FUNC SIMILAR(rows, column$, cutoff)
    DIM header = rows[0]
    DIM col = -1
    DIM r = 0
    FOR r = 0 TO LEN(header) - 1
        IF header[r] = column$ THEN col = r
    NEXT r
    DIM out = []
    IF col < 0 THEN RETURN out
    DIM cells = SELECT(LAMBDA USE(col) row -> "" + row[col], _
        DROP(1, rows))
    DIM names = UNIQUE(FILTER(LAMBDA v$ -> v$ <> "", cells))
    DIM i = 0
    DIM j = 0
    FOR i = 0 TO LEN(names) - 1
        FOR j = i + 1 TO LEN(names) - 1
            DIM a$ = names[i]
            DIM b$ = names[j]
            DIM score = FUZZY.SCORE(a$, b$, "token_sort", 1)
            IF score >= cutoff THEN PUSH(out, [a$, b$, ROUND(score)])
        NEXT j
    NEXT i
    RETURN out
ENDFUNC

' The fields of one CSV line, with quoted fields that hold the
' separator or doubled quotes.
EXPORT FUNC CSV_FIELDS(line$, sep$)
    DIM out = []
    DIM cur$ = ""
    DIM quoted = FALSE
    DIM q$ = CHR$(34)
    DIM chars = -line$

```

```

DIM k = 0
DO WHILE k < LEN(chars)
  DIM ch$ = chars[k]
  IF quoted THEN
    DIM after$ = ""
    IF k + 1 < LEN(chars) THEN after$ = chars[k + 1]
    IF ch$ = q$ AND ALSO after$ = q$ THEN
      cur$ = cur$ + q$
      k = k + 1
    ELSEIF ch$ = q$ THEN
      quoted = FALSE
    ELSE
      cur$ = cur$ + ch$
    ENDIF
  ELSEIF ch$ = q$ THEN
    quoted = TRUE
  ELSEIF ch$ = sep$ THEN
    PUSH(out, cur$)
    cur$ = ""
  ELSE
    cur$ = cur$ + ch$
  ENDIF
  k = k + 1
LOOP
PUSH(out, cur$)
RETURN out
ENDFUNC

' The rows of a CSV file or of the first sheet of a workbook.
EXPORT FUNC READ_TABLE(path$, sep$)
  IF LCASE$(PATH.EXT$(path$)) = ".xlsx" THEN
    DIM sheets = XLSX.SHEETS(path$)
    DIM book = XLSX.READ(path$)
    RETURN book{sheets[0]}
  ENDIF
  DIM text$ = REPLACE$(TXTREADER$(path$), CHR$(13), "")
  DIM lines = SPLIT(text$, CHR$(10))
  DIM rows = []
  DIM k = 0
  FOR k = 0 TO LEN(lines) - 1
    DIM one$ = lines[k]
    IF TRIM$(one$) <> "" THEN PUSH(rows, CSV_FIELDS(one$, sep$))
  NEXT k
  RETURN rows
ENDFUNC

' A header row followed by the rows of a list.
FUNC WithHeader(head, rows)

```



```

    DIM out = [head]
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM one = rows[k]
        PUSH(out, one)
    NEXT k
    RETURN out
ENDFUNC

' The result as a workbook: Clean, Problems and Similar names.
EXPORT FUNC WRITE_XLSX(result, similar, path$)
    DIM wb = XLSX.NEW()
    DIM clean = XLSX.SHEET(wb, "Clean", result{"rows"})
    XLSX.FREEZE(clean, 1)
    DIM head = ["Row", "Column", "Value", "Problem"]
    DIM prob = WithHeader(head, result{"problems"})
    XLSX.SHEET(wb, "Problems", prob)
    DIM sim = WithHeader(["Name", "Looks like", "Score"], similar)
    XLSX.SHEET(wb, "Similar names", sim)
    RETURN XLSX.WRITE(path$, wb)
ENDFUNC

```

How it works

1. **Reading the export.** `READ_TABLE` reads a workbook with `XLSX.READ` and a CSV file with its own small parser, `CSV_FIELDS`. The parser exists because of one common case: a customer called *Weber; Sohn & Co* in a file separated by semicolons. The export puts such a field in quotes, and a plain split at every semicolon would cut the name in two. `CSV_FIELDS` walks the line character by character and remembers whether it is inside quotes. Two quotes in a row inside a quoted field stand for one quote character. `CSVREADER` handles quotes as well, but it turns a cell that looks like a number into a number. The cleaner wants every cell exactly as written, a postcode with its leading zero included, so it reads the text itself.
1. **One kind per column.** `GUESS_KINDS` looks at the column names: a name with *dat* in it is a date, *Amount* or *Betrag* a number, and so on. A `[data_cleaner.columns]` table in `work.conf` can overrule the guess for any column. Every column the cleaner does not know stays text, which means its spaces are tidied and nothing else.
1. **Dates.** `ISO_DATE$` tries four regular expressions, one per form. ISO dates and dates with dots are clear: the dot is the German form, always day first.

A slash is the hard case. *05/10/2026* is the 5th of October in Europe and the 10th of May in the United States, and no program can tell from the text alone. That is the `day_first` setting. Month names are compared by their first three letters, so *Oct*, *Oct.* and *October* all work. `Ymd$` finally asks `DT.DAYS_IN` how many days the month has, which is how the 31st of February is caught. A workbook can also hold a date as a number, the days since the 30th of December 1899; `ISO_DATE$` turns that into a date too.

1. **Numbers.** `NUMBER` removes spaces and currency signs, turns brackets and a minus behind into a minus, and then decides which separator is the decimal one. With `decimal = "auto"` the last separator wins: in *1.234,50* the comma comes last, so it is the decimal comma and the point separates thousands. Before and after that step a regular expression checks the shape of the text. This check matters, since `VAL("12abc")` answers 12 without a complaint, and a cleaner that turns *12abc* into 12 hides the problem it should report.
1. **Problems instead of guesses.** `CELL` answers a map with the clean value and a problem text. When the problem text is not empty, the cell keeps its old value and `CLEAN` adds a row to the problem list: the row number as Excel shows it, the column, the value and the reason. The cleaner never invents a value. A wrong IBAN stays wrong, but now it is on a list.
1. **Repeated rows.** `CLEAN` builds a key from the clean values of each row and keeps a map of the keys it has seen. A row whose key is already in the map is dropped. Comparing the clean values is the point: *Müller GmbH* with *1.234,50* and *Müller GmbH* with *1234.50* are the same row once both are cleaned.
1. **Names that look alike.** `SIMILAR` takes the cells of the name column with a `SELECT`, drops the empty ones with a `FILTER` and keeps each name once with `UNIQUE`. Then it compares every name with every other one using `FUZZY.SCORE` from the fuzzy library. The method `token_sort` sorts the words first, so *GmbH Müller* and *Müller GmbH* score 100. *Müller GmbH* and *Mueller GmbH* score 87, above the cutoff of 85. The recipe only lists such pairs; deciding whether they are one customer is your job.

Run it

With the example export of the recipe's folder:

```
jdbasic data_cleaner.jdb
Rows read: 5
Cells changed: 11
Repeated rows dropped: 1
Problems: 3
  row 4, IBAN: not a valid IBAN (DE00 1234)
  row 4, Mail: not a mail address (weber@example)
  row 6, Date: not a date (31.02.2026)
Names that look alike: 1
  Müller GmbH / Mueller GmbH (87)
Clean workbook: C:/Data/orders_clean.xlsx
```

The clean workbook is written next to the export, with `_clean` added to its name. With `--dry-run` the recipe prints the summary and writes nothing.

Schedule it

The wizard plans the cleaner for Monday at 7:30. Point the export of the shop system at the same file name every week, and the clean workbook is ready before the first coffee. If the export arrives at other times, let the Hot Folder recipe (M10) start the cleaner whenever a new file lands.

Make it yours

The settings are in the part of `work.conf` that starts with `[data_cleaner]`:

```
[data_cleaner]
input = "~/Documents/AutomateWork/export.csv"
separator = ";"
day_first = true
name_column = "Customer"
similar = 85

[data_cleaner.columns]
Ordered = "date"
Net = "number"
"Customer no" = "keep"
```

The columns table names the kind of a column when the guess from its name is wrong. keep leaves a column exactly as it is, which is right for article numbers with leading zeros.

The interesting changes are in the code:

- **A different date form.** Some systems write 20261005. Add one more test to ISO_DATE\$, right before its final RETURN "":

```
g = REGEX.FINDALL("^[0-9]{4})([0-9]{2})([0-9]{2})$", s$)
IF LEN(g) > 0 THEN
    RETURN Ymd$(VAL(g[0][0]), VAL(g[0][1]), VAL(g[0][2]))
ENDIF
```

- **A new kind of column.** A column of German postcodes can get its own kind. Add a block to CELL before the last line:

```
IF kind$ = "postcode" THEN
    IF LEN(REGEX.FINDALL("[0-9]{5}$", t$)) = 0 THEN
        RETURN Answer(t$, "not a postcode")
    ENDIF
ENDIF
```

and name the column in [data_cleaner.columns] as PLZ = "postcode".

- **Repeated rows by a key column.** Two rows with the same order number are the same order, even if the amount was corrected. In CLEAN, replace the line that adds to key\$ with IF col\$ = "Order" THEN key\$ = now\$. The key is then the order number alone, and the second row of each order drops out.

When it goes wrong

- **Every row is one long field:** the separator is wrong. Open the file in a text editor and set separator to the character between the fields.
- **Dates are off by months:** day_first does not match the export. Look at a date with a day above 12; if 13/10 turns up as a problem, switch the setting.
- **Too many similar names:** raise similar to 90. Names like *Weber AG* and *Weber KG* are different companies that differ in one letter, and a lower cut-off will pair them.

- **Leading zeros are gone:** the column was guessed as a number. Set it to keep in `[data_cleaner.columns]`.

Balance dividend

About 30 minutes of repair work every Monday, and the bank's returned payment found on Monday morning instead of two weeks later.

M12 Meeting Cost Meter

The chore

Jonas leads a team of six. His calendar is full, and every quarter someone asks him why the project is late. He suspects the answer is in his calendar: the weekly call that once took thirty minutes and now takes an hour, the status meeting with eight people where two would do, the review that repeats every second Tuesday although nobody remembers why. He has never added it up. Adding up a month of calendar entries by hand takes an evening, and by the time it is done the next month has started.

What you get

Every Friday afternoon the Meeting Cost Meter reads your calendar for the last four weeks and answers three questions. How many hours went into meetings, week by week and month by month? What did that time cost, your own time and the time of everyone in the room? Which meeting series take the most of it? The answer is printed as a short report and saved as a workbook with one sheet for the weeks, one for the months and one for the series, so you can sort, filter and put it in front of whoever asked.

Before you start

The recipe reads a calendar file in the iCalendar format (`.ics`). Every calendar program can write one:

- **Outlook:** *File, Save Calendar*, choose a date range of at least the last month and *Full details*.
- **Google Calendar:** *Settings, Import and export, Export*.
- **A shared calendar with a link:** download the `.ics` file from the link.

Save the file in your work folder as `calendar.ics`, or tell the recipe where it is. Chapter 5 shows how to fetch it automatically from the calendar server; for now, a fresh export once a month is enough.

The program

The program reads the settings, picks the weeks to look at, and leaves the counting to the module:

```
' Meeting Cost Meter: how many hours of the last weeks went into
' meetings, what they cost, and which series take the most. The
' settings are the [meeting_cost] part of work.conf.
,
'   jdbasic meeting_cost.jdb           report and workbook
'   jdbasic meeting_cost.jdb --dry-run  the report only

IMPORT WORKCONF, MEETCOST, ICAL, DT

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "meeting_cost."
DIM cal_file$ = WORKCONF.FOLDER$(cfg, key$ + "calendar", "")
DIM rate = WORKCONF.VALUE(cfg, key$ + "hourly_rate", 60)
DIM currency$ = WORKCONF.VALUE(cfg, key$ + "currency", "EUR")
DIM weeks_back = WORKCONF.VALUE(cfg, key$ + "weeks", 4)
DIM top_n = WORKCONF.VALUE(cfg, key$ + "top", 5)
DIM book$ = "~/Documents/AutomateWork/meeting_cost.xlsx"
DIM out$ = WORKCONF.FOLDER$(cfg, key$ + "workbook", book$)
IF cal_file$ = "" THEN THROW "Set meeting_cost.calendar first."

DIM offset = DT.LOCAL_OFFSET()
DIM now = DT.NOW()
DIM back$ = "-" + STR$(weeks_back) + " weeks"
DIM from = DT.STARTOF(DT.ADD(now, back$, offset), "week", offset)
DIM cal = ICAL.PARSEFILE(cal_file$)
DIM meetings = MEETCOST.MEETINGS(cal, from, now)
DIM sum = MEETCOST.SUMMARY(meetings, offset)

PRINT "Since "; DT.FMT$(from, "%Y-%m-%d", offset)
PRINT MEETCOST.REPORT$(sum, rate, currency$, top_n)
IF NOT WORKCONF.FLAG("dry-run") THEN
    MKDIR PATH.DIRNAME$(out$)
    MEETCOST.WRITE_XLSX(sum, rate, out$)
    PRINT
    PRINT "Workbook: "; out$
ENDIF
```

The module turns calendar entries into hours and the hours into a report and a workbook:

```

' meetcost.jdb: the time a calendar spends in meetings. Hours and cost
' per week and per month, and the meeting series that take the most.
'
'   IMPORT MEETCOST, ICAL, DT
'   DIM cal = ICAL.PARSEFILE("calendar.ics")
'   DIM list = MEETCOST.MEETINGS(cal, from, upto)
'   DIM sum = MEETCOST.SUMMARY(list, 2)
'   PRINT MEETCOST.REPORT$(sum, 60, "EUR", 5)

EXPORT MODULE MEETCOST

IMPORT ICAL, DT, DF, XLSX

' Every meeting in [from, upto) as a map with "summary", "start",
' "hours" and "people" (the attendees, at least 1). All-day events are
' left out: they block a day, not a meeting room.
EXPORT FUNC MEETINGS(cal, from, upto)
    DIM out = []
    DIM frame = ICAL.EXPAND(cal, from, upto)
    DIM k = 0
    FOR k = 0 TO DF.NROWS(frame) - 1
        DIM r = DF.ROW(frame, k)
        DIM ev = r{"event"}
        IF NOT ICAL.ALLDAY(cal, ev) THEN
            DIM people = LEN(ICAL.VALUES(cal, ev, "ATTENDEE"))
            IF people < 1 THEN people = 1
            DIM hours = (r{"end"} - r{"start"}) / 3600
            PUSH(out, {"summary": r{"summary"}, "start": r{"start"}, _
                "hours": hours, "people": people})
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' "2026-W41" for the ISO week of an instant.
EXPORT FUNC WEEK$(t, offset)
    DIM w = DT.ISOWEEK(t, offset)
    RETURN STR$(w{"year"}) + "-W" + RIGHT$("0" + STR$(w{"week"}), 2)
ENDFUNC

SUB AddTo(totals, key$, hours)
    IF NOT MAP.EXISTS(totals, key$) THEN totals[key$] = 0
    totals[key$] = totals[key$] + hours
ENDSUB

' The totals of a list of meetings: "hours" in all, "person_hours"
' (hours times the people in the room), "weeks" and "months" (maps of
' hours), and "series" (a map per title with "hours", "times" and

```



```

' "person_hours").
EXPORT FUNC SUMMARY(meetings, offset)
  DIM weeks AS MAP
  DIM months AS MAP
  DIM series AS MAP
  DIM hours = 0
  DIM person_hours = 0
  DIM k = 0
  FOR k = 0 TO LEN(meetings) - 1
    DIM m = meetings[k]
    DIM h = m{"hours"}
    DIM ph = h * m{"people"}
    hours = hours + h
    person_hours = person_hours + ph
    AddTo(weeks, WEEK$(m{"start"}, offset), h)
    AddTo(months, DT.FMT$(m{"start"}, "%Y-%m", offset), h)
    DIM title$ = m{"summary"}
    IF NOT MAP.EXISTS(series, title$) THEN
      series{title$} = {"hours": 0, "times": 0, _
        "person_hours": 0}
    ENDIF
    DIM s = series{title$}
    s{"hours"} = s{"hours"} + h
    s{"times"} = s{"times"} + 1
    s{"person_hours"} = s{"person_hours"} + ph
  NEXT k
  RETURN {"hours": hours, "person_hours": person_hours, _
    "weeks": weeks, "months": months, "series": series}
ENDFUNC

' The series titles, the one with the most hours first.
EXPORT FUNC TOP_SERIES(summary)
  DIM series = summary{"series"}
  DIM titles = MAP.KEYS(series)
  DIM hours = SELECT(LAMBDA USE(series) t$ -> series{t$}{"hours"}, _
    titles)
  RETURN SELECT(LAMBDA USE(titles) i -> titles[i], GRADE(-hours))
ENDFUNC

FUNC Hours$(h)
  RETURN FORMAT$("{:>6.1f} h", h)
ENDFUNC

FUNC Money$(v, currency$)
  RETURN FORMAT$("{:.0f}", v) + " " + currency$
ENDFUNC

' The report as text: the totals, the hours per week and month, and

```

```

' the top_n series with the most hours.
EXPORT FUNC REPORT$(summary, rate, currency$, top_n)
    DIM out = []
    DIM h = summary{"hours"}
    PUSH(out, "Meetings: " + FORMAT$("{:.1f}", h) + " hours, " + _
        Money$(h * rate, currency$) + " of your time")
    PUSH(out, "Everyone in them: " + _
        FORMAT$("{:.1f}", summary{"person_hours"}) + " hours, " + _
        Money$(summary{"person_hours"} * rate, currency$))
    PUSH(out, "")
    PUSH(out, "By week")
    DIM weeks = summary{"weeks"}
    out = APPEND(out, SELECT(LAMBDA USE(weeks) w$ -> _
        " " + w$ + " " + Hours$(weeks{w$}), SORT(MAP.KEYS(weeks))))
    PUSH(out, "")
    PUSH(out, "By month")
    DIM months = summary{"months"}
    out = APPEND(out, SELECT(LAMBDA USE(months) m$ -> _
        " " + m$ + " " + Hours$(months{m$}), _
        SORT(MAP.KEYS(months))))
    PUSH(out, "")
    PUSH(out, "Largest series")
    DIM top = TAKE(top_n, TOP_SERIES(summary))
    DIM k = 0
    FOR k = 0 TO LEN(top) - 1
        DIM t$ = top[k]
        DIM s = summary{"series"}{t$}
        DIM times$ = STR$(s{"times"}) + " times"
        IF s{"times"} = 1 THEN times$ = "once"
        PUSH(out, " " + Hours$(s{"hours"}) + " " + t$ + _
            " (" + times$ + ")")
    NEXT k
    RETURN JOIN(out, CHR$(10))
ENDFUNC

' The summary as a workbook with the sheets Weeks, Months and Series.
EXPORT FUNC WRITE_XLSX(summary, rate, path$)
    DIM wb = XLSX.NEW()
    DIM rows = [{"Week", "Hours", "Cost"}]
    DIM weeks = summary{"weeks"}
    DIM keys = SORT(MAP.KEYS(weeks))
    DIM k = 0
    FOR k = 0 TO LEN(keys) - 1
        DIM wh = weeks{keys[k]}
        PUSH(rows, [keys[k], wh, wh * rate])
    NEXT k
    XLSX.SHEET(wb, "Weeks", rows)
    rows = [{"Month", "Hours", "Cost"}]

```

```

DIM months = summary{"months"}
keys = SORT(MAP.KEYS(months))
FOR k = 0 TO LEN(keys) - 1
    DIM mh = months[keys[k]]
    PUSH(rows, [keys[k], mh, mh * rate])
NEXT k
XLSX.SHEET(wb, "Months", rows)
rows = [{"Series", "Times", "Hours", "Person hours", "Cost"}]
DIM top = TOP_SERIES(summary)
FOR k = 0 TO LEN(top) - 1
    DIM t$ = top[k]
    DIM s = summary{"series"}{t$}
    PUSH(rows, [t$, s{"times"}, s{"hours"}, s{"person_hours"}, _
        s{"person_hours"} * rate])
NEXT k
DIM sheet = XLSX.SHEET(wb, "Series", rows)
XLSX.COLUMN(sheet, "A", {"width": 30})
RETURN XLSX.WRITE(path$, wb)
ENDFUNC

```

How it works

The recipe is built from four small steps, each in a function of its own. Reading them in order is a good way to see how a medium recipe is put together.

1. **From a file to meetings.** `ICAL.PARSEFILE` reads the calendar, and `ICAL.EXPAND` turns it into one row per meeting in the time range. This is the important part: a weekly call is stored in the file once, with a rule that says *every week*. `EXPAND` applies the rule, so a call that met four times becomes four rows. `MEETINGS` then keeps what it needs from each row: the title, the start, the length in hours, and the number of people invited. All-day entries, such as a public holiday or an offsite, are left out, since they are not meetings in the sense of this recipe.
1. **From meetings to totals.** `SUMMARY` walks the list once and adds every meeting to three maps at the same time: one keyed by week, one by month and one by title. A map is the natural tool here: the key is the label you want in the report, and `AddTo` creates the entry the first time a key turns up. *Person hours* are the hours of a meeting times the people in it; an hour with six people costs six hours of work.
1. **Weeks the way calendars count them.** `WEEK$` uses `DT.ISOWEEK`, the ISO 8601 week that European calendars print: weeks start on Monday, and week

1 is the week with the first Thursday of the year. That is why 1 January 2027 belongs to week 53 of 2026. Counting weeks yourself goes wrong at exactly these edges, so leave it to the library.

1. **From totals to answers.** TOP_SERIES takes the hours of each title with a SELECT and sorts the titles by them. It does that with GRADE, which answers the order in which the values would be sorted; negating the hours turns *smallest first* into *largest first*, and a second SELECT turns the positions back into titles. REPORT\$ and WRITE_XLSX then present the same summary twice, once as text and once as a workbook. REPORT\$ makes the lines per week and per month with a SELECT each, and TAKE keeps the first top series.

Each function takes what it needs as parameters and returns its result. None of them reads settings or prints anything. That is what makes the module testable: the test gives it a calendar file and a fixed range and checks the numbers.

Run it

```
jdbasic meeting_cost.jdb
Since 2026-09-28
Meetings: 7.5 hours, 450 EUR of your time
Everyone in them: 23.0 hours, 1380 EUR

By week
  2026-W41      2.5 h
  2026-W42      1.0 h
  2026-W43      3.0 h
  2026-W44      1.0 h

By month
  2026-10      7.5 h

Largest series
  4.0 h Weekly team call (4 times)
  2.0 h Budget review (once)
  1.5 h Miller project status (once)

Workbook: C:\Users\jonas\Documents\AutomateWork\meeting_cost.xlsx
```

This is the small calendar the recipe's test uses. On a real calendar the list of series is longer, and the weekly rows show at a glance which weeks were the full ones. With --dry-run the recipe prints the report and writes no workbook.

Schedule it

The wizard plans the meter for Friday at 15:00, which leaves time to look at the numbers before the next week is planned. Remember to export a fresh calendar file before, or let Chapter 5 do that for you.

Make it yours

The settings are in the part of `work.conf` that starts with `[meeting_cost]`:

```
[meeting_cost]
calendar = "~/Documents/AutomateWork/calendar.ics"
hourly_rate = 60
currency = "EUR"
weeks = 4
top = 5
workbook = "~/Documents/AutomateWork/meeting_cost.xlsx"
```

`hourly_rate` is the cost of an hour of work. Ask your controlling team for the rate they use. Without one, pick a round figure you can defend and say so when you show the numbers; the point is the comparison between weeks and series, not the exact amount.

Beyond the settings, three changes in the code are worth knowing:

- **Leave out meetings that are not meetings.** Focus time blocked in the calendar is counted as a meeting. To skip every entry whose title starts with Focus, add one condition in MEETINGS, right after `IF NOT ICAL.ALLDAY(cal, ev)` THEN:

```
IF STARTSWITH(r{"summary"}, "Focus") THEN CONTINUEFOR
```

- **Count only meetings with others.** Change the line `IF people < 1 THEN people = 1` into `IF people < 2 THEN CONTINUEFOR`. Entries with nobody invited, the reminders you set for yourself, then drop out.
- **Group by person instead of by title.** SUMMARY adds every meeting to a map keyed by its title. Use the first attendee instead (`ICAL.VALUES(cal, ev, "ATTENDEE")[0]`, kept in MEETINGS under a new key), and the series list becomes a list of the people you spend the most meeting time with.

When it goes wrong

- **"Set meeting_cost.calendar first":** the wizard did not know where your calendar file is. Enter the path in `work.conf`.
- **No meetings at all:** the export covered a different range. Check that the file reaches back the number of weeks the recipe looks at.
- **A series is counted under two titles:** someone renamed it. The recipe groups by title, so "Team call" and "Weekly team call" are two series. Rename them in the calendar or in the workbook.
- **Times are an hour off:** the export uses a time zone the recipe reads as UTC. Export again with the time zone included (Outlook does this with *Full details*).

Balance dividend

About 15 minutes a week of looking at the calendar, and the first honest answer to the question where the team's time goes.

M13 Snippet Tool

The chore

Jonas writes the same few texts all day. The answer to a supplier who sent the wrong invoice. The note that a customer's request has arrived and will be answered by Friday. The polite refusal of a meeting. Each time he opens an old mail, copies the text, changes the name and the date, and forgets one of them about once a week. The texts live in his sent folder, and finding the right one takes longer than writing it.

What you get

A small command that keeps such texts as files in one folder and fills them in when you need them. A text holds placeholders in double braces, such as `{{name}}` or `{{due}}`. `copy` fills them from the command line and puts the result on the clipboard, ready to paste into a mail, a chat or a form. `show` prints it instead, `list` shows what is there and which placeholders each text expects, and `add` saves a new one. `{{today}}` is always filled with the date of the day.

Before you start

The setup wizard has run. The snippets live in a folder of their own, snippets in your work folder unless you choose another one. Each snippet is a plain text file there: `late.txt` holds the snippet `late`. You can write them with `add` or with Notepad, whichever is quicker; the tool does not mind.

The program

The program describes its four commands, reads the settings and does what the command says:

```
' Snippet Tool: the text blocks you type ten times a day, kept as
' files and filled in on the command line. The settings are the
' [snippet_tool] part of work.conf.
'
' jdbasic snippet_tool.jdb list
' jdbasic snippet_tool.jdb show thanks --set name=Ann
' jdbasic snippet_tool.jdb copy thanks --set name=Ann
' jdbasic snippet_tool.jdb add thanks --text "Dear {{name}}, ..."
```

```

IMPORT WORKCONF, SNIPPETS, CLI

DIM spec = CLI.NEW("snippet_tool", "text blocks with placeholders")
DIM c_list = CLI.CMD(spec, "list", "the snippets and their places")
DIM c_show = CLI.CMD(spec, "show", "print a snippet filled in")
CLI.ARG(c_show, "snippet", "the snippet's name", TRUE)
CLI.MANY(c_show, "set", "s", "a value, as name=value")
DIM c_copy = CLI.CMD(spec, "copy", "a filled snippet to clipboard")
CLI.ARG(c_copy, "snippet", "the snippet's name", TRUE)
CLI.MANY(c_copy, "set", "s", "a value, as name=value")
DIM c_add = CLI.CMD(spec, "add", "save a new snippet")
CLI.ARG(c_add, "snippet", "the snippet's name", TRUE)
CLI.OPT(c_add, "text", "t", "the text; \n starts a new line", "")
CLI.FLAG(c_add, "force", "f", "replace a snippet of that name")

DIM cmds = [c_list, c_show, c_copy, c_add]
DIM ci = 0
FOR ci = 0 TO LEN(cmds) - 1
    CLI.FLAG(cmds[ci], "dry-run", "", "show what would change")
    CLI.OPT(cmds[ci], "config", "", "the settings file", "")
NEXT ci

DIM argv = APPEND(["snippet_tool.exe"], DROP(1, OS.ARGS()))
DIM got = CLI.PARSE(spec, argv)
IF CLI.DONE(spec, got) THEN END CLI.STATUS(got)

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "snippet_tool."
DIM home$ = "~/Documents/AutomateWork/snippets"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM empty AS MAP
DIM standing = WORKCONF.VALUE(cfg, key$ + "values", empty)
DIM today$ = FORMAT_DATE(NOW(), "%d %B %Y")
DIM cmd$ = got{"command"}

IF cmd$ = "list" THEN
    DIM names = SNIPPETS.LIST(folder$)
    IF LEN(names) = 0 THEN PRINT "No snippets yet in "; folder$
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM t$ = SNIPPETS.READ$(folder$, names[k])
        DIM places = SNIPPETS.PLACEHOLDERS(t$)
        PRINT names[k]; " "; JOIN(places, ", ")
    NEXT k
ELSEIF cmd$ = "show" ORELSE cmd$ = "copy" THEN
    DIM snip$ = got{"args"}[0]
    DIM given = SNIPPETS.VALUES(got{"set"})

```



```

    DIM values = SNIPPETS.MERGE(standing, today$, given)
    DIM filled = SNIPPETS.FILL(SNIPPETS.READ$(folder$, snip$), values)
    IF cmd$ = "copy" AND ALSO NOT got{"dry-run"} THEN
        CLIPBOARD.SET filled{"text"}
        PRINT "On the clipboard: "; snip$
    ELSE
        PRINT filled{"text"}
    ENDIF
    IF LEN(filled{"missing"}) > 0 THEN
        PRINT "Still open: "; JOIN(filled{"missing"}, ", ")
    ENDIF
ELSEIF cmd$ = "add" THEN
    DIM new_name$ = got{"args"}[0]
    DIM body$ = REPLACE$(got{"text"}, "\n", CHR$(10))
    IF got{"dry-run"} THEN
        PRINT "would save "; new_name$; ":"
        PRINT body$
    ELSE
        TRY
            SNIPPETS.WRITE(folder$, new_name$, body$, got{"force"})
            PRINT "Saved "; new_name$
        CATCH
            PRINT ERRMSG$
        ENDTRY
    ENDIF
ELSE
    PRINT CLI.HELP$(spec)
ENDIF

```

The module knows the folder and the placeholders:

```

' snippets.jdb: text blocks kept as files in a folder, a file for
' each snippet. A snippet may hold placeholders such as {{name}}
' that are filled when it is used.
'
'   IMPORT SNIPPETS
'   DIM t$ = SNIPPETS.READ$("C:/snips", "thanks")
'   DIM filled = SNIPPETS.FILL(t$, {"name": "Ann"})
'   PRINT filled{"text"}

EXPORT MODULE SNIPPETS

' TRUE for a name that can be a snippet: letters, digits, _ and -.
EXPORT FUNC VALID_NAME(snip$)
    IF snip$ = "" OR ELSE LEN(snip$) > 40 THEN RETURN FALSE
    RETURN LEN(REGEX.FINDALL("[A-Za-z0-9_-]+$", snip$)) > 0
ENDFUNC

```

```

FUNC FileOf$(folder$, snip$)
    IF NOT VALID_NAME(snip$) THEN THROW "Not a snippet name: " + snip$
    RETURN PATH.JOIN$(folder$, LCASE$(snip$) + ".txt")
ENDFUNC

' The names of all snippets in the folder, sorted.
EXPORT FUNC LIST(folder$)
    IF NOT FILE.ISDIR(folder$) THEN RETURN []
    DIM files = DIR$(PATH.JOIN$(folder$, "*.txt"))
    RETURN SORT(SELECT(LAMBDA f$ -> LEFT$(f$, LEN(f$) - 4), files))
ENDFUNC

' The text of a snippet, line ends as LF.
EXPORT FUNC READ$(folder$, snip$)
    DIM path$ = FileOf$(folder$, snip$)
    IF NOT FILE.EXISTS(path$) THEN THROW "No snippet called " + snip$
    RETURN REPLACE$(TXTREADER$(path$), CHR$(13), "")
ENDFUNC

' Saves a snippet; an existing one is replaced only with replace.
EXPORT SUB WRITE(folder$, snip$, txt$, replace)
    DIM path$ = FileOf$(folder$, snip$)
    IF FILE.EXISTS(path$) AND ALSO NOT replace THEN
        THROW "A snippet called " + snip$ + " exists; use --force"
    ENDIF
    MKDIR folder$
    TXTWRITER path$, txt$
ENDSUB

' The placeholder names of a text, each once, in order.
EXPORT FUNC PLACEHOLDERS(txt$)
    DIM found = REGEX.FINDALL("\{\{ *( [A-Za-z0-9_ ]+ ) *\}\}\}", txt$)
    RETURN UNIQUE(SELECT(LAMBDA hit -> LCASE$(hit[0]), found))
ENDFUNC

' Values from settings like "name=Ann": a map, names in small letters.
EXPORT FUNC VALUES(pairs)
    DIM out AS MAP
    DIM k = 0
    FOR k = 0 TO LEN(pairs) - 1
        DIM pair$ = pairs[k]
        DIM at = INSTR(pair$, "=")
        IF at <= 0 THEN THROW "Use name=value, not " + pair$
        DIM key$ = LCASE$(TRIM$(LEFT$(pair$, at)))
        out{key$} = MID$(pair$, at + 1, LEN(pair$))
    NEXT k
    RETURN out

```

```
ENDFUNC
```

```
' The text with every placeholder that has a value replaced. Answers
' "text" and "missing", the names that had no value.
```

```
EXPORT FUNC FILL(txt$, values)
```

```
    DIM out$ = ""
```

```
    DIM missing = []
```

```
    DIM rest$ = txt$
```

```
    DO
```

```
        DIM open_at = INSTR(rest$, "{{")
```

```
        DIM close_at = -1
```

```
        IF open_at >= 0 THEN close_at = INSTR(open_at, rest$, "}}")
```

```
        IF close_at < 0 THEN EXITDO
```

```
        DIM inner$ = MID$(rest$, open_at + 2, close_at - open_at - 2)
```

```
        DIM key$ = LCASE$(TRIM$(inner$))
```

```
        out$ = out$ + LEFT$(rest$, open_at)
```

```
        IF MAP.EXISTS(values, key$) THEN
```

```
            DIM value$ = "" + values{key$}
```

```
            out$ = out$ + value$
```

```
        ELSE
```

```
            out$ = out$ + "{" + inner$ + "}"
```

```
            IF NOT (key$ IN missing) THEN PUSH(missing, key$)
```

```
        ENDIF
```

```
        rest$ = MID$(rest$, close_at + 2, LEN(rest$))
```

```
    LOOP
```

```
    RETURN {"text": out$ + rest$, "missing": missing}
```

```
ENDFUNC
```

```
' The values for a snippet: the standing ones from the settings, then
```

```
' today's date, then the ones from the command line, each later one
```

```
' winning over an earlier one.
```

```
EXPORT FUNC MERGE(standing, today$, given)
```

```
    DIM out AS MAP
```

```
    DIM keys = MAP.KEYS(standing)
```

```
    DIM k = 0
```

```
    FOR k = 0 TO LEN(keys) - 1
```

```
        out{LCASE$(keys[k])} = standing{keys[k]}
```

```
    NEXT k
```

```
    out{"today"} = today$
```

```
    keys = MAP.KEYS(given)
```

```
    FOR k = 0 TO LEN(keys) - 1
```

```
        out{keys[k]} = given{keys[k]}
```

```
    NEXT k
```

```
    RETURN out
```

```
ENDFUNC
```

How it works

1. **Four commands, one description.** `CLI.NEW` starts the description of the program, and `CLI.CMD` adds a command with its own arguments and options. `CLI.MANY` declares an option you may give more than once: `--set name=Ann --set due=Friday`, or `-s` for short. The loop after the commands adds `--dry-run` and `--config` to all four of them. `CLI.PARSE` then checks the command line against the description and answers a map with the command, its arguments and its options. `CLI.DONE` handles `--help` and every mistake, such as a missing name, by printing the usage and ending the program.
1. **One file per snippet.** `FileOf$` turns a name into a file name in the folder, always in small letters, so `Late` and `late` are the same snippet. `VALID_NAME` allows letters, digits, `_` and `-` only. That keeps a name from reaching outside the folder: `.. \boot` is no snippet name. `WRITE` refuses to overwrite a snippet unless you pass `--force`, so a typo in `add` does not cost you a text you spent care on.
1. **Finding the placeholders.** `PLACEHOLDERS` asks `REGEX.FINDALL` for every `{{name}}` in the text. The pattern allows spaces inside the braces, so `{{ name }}` counts as well. A `SELECT` takes the name out of each hit in small letters, and `UNIQUE` keeps each name once, in the order it first appears, which is the order `list` shows.
1. **Filling them in.** `FILL` walks through the text from one `{{` to the next `}}` and builds the result piece by piece. A placeholder with a value is replaced by it. One without a value stays in the text exactly as it was written, and its name is added to `missing`, so the program can tell you `Still open: due`. A text with a stray `{{` and no closing braces is left alone.
1. **Where the values come from.** `MERGE` puts three sources together, each one able to override the one before: the standing values from `work.conf` (your name, your phone number), then `today`, then whatever you gave with `--set`. `VALUES` turns the `--set` pairs into a map; everything after the first `=` belongs to the value, so `--set link=https://example.com/?a=1` works.

The clipboard is touched in one place only, the line with `CLIPBOARD.SET` in the program. The module never sees it, and neither does the test.

Run it

Jonas keeps the note he sends when a meeting runs over in the file `late.txt` in the snippets folder:

```
Hello {{name}},
I will be {{minutes}} minutes late.
{{sign}}
```

He adds a short one from the command line, looks at what is there, and fills the first one in:

```
jdbasic snippet_tool.jdb add thanks --text "Thank you, {{name}}."
Saved thanks

jdbasic snippet_tool.jdb list
late  name, minutes, sign
thanks name

jdbasic snippet_tool.jdb show late -s name=Anna
Hello Anna,
I will be {{minutes}} minutes late.
Jonas
Still open: minutes

jdbasic snippet_tool.jdb copy late -s name=Anna -s minutes=10
On the clipboard: late
```

In `--text`, `\n` starts a new line. `sign` came from the standing values in `work.conf` (see below). With `--dry-run`, `copy` prints the text and leaves the clipboard alone, and `add` shows what it would save.

Schedule it

There is nothing to schedule: the tool runs when you call it. What helps is a shorter way to call it. Save these lines as `snip.cmd` in a folder on your `PATH`, such as the `jdBasic` folder:

```
@echo off
setlocal
set HERE=%USERPROFILE%\Documents\AutomateWork\recipes
jdbasic "%HERE%\M13_snippet_tool\snippet_tool.jdb" %*
```

From then on `snip copy late -s name=Anna` does the same from any command prompt, and the Windows *Run* box (Windows key and R) accepts it as well.

Make it yours

The settings are in the part of `work.conf` that starts with `[snippet_tool]`. The standing values go into a table of their own below it:

```
[snippet_tool]
folder = "~/Documents/AutomateWork/snippets"

[snippet_tool.values]
sign = "Jonas"
phone = "+49 421 555 0100"
team = "Purchasing"
```

Point `folder` at a shared drive, and the whole team uses the same texts. Everyone keeps their own standing values, so `{{sign}}` signs with the name of whoever copies.

Three changes in the code are worth knowing:

- **A different date.** `{{today}}` is written as 02 October 2026. For 02.10.2026, change the format in the line that sets `today$`:

```
DIM today$ = FORMAT_DATE(NOW(), "%d.%m.%Y")
```

- **A placeholder for a deadline.** Many answers promise something by a date. Add a value right after the line with `SNIPPETS.MERGE`, and `{{in_a_week}}` is filled with the day a week from now:

```
DIM week = DATEADD("D", 7, NOW())
values{"in_a_week"} = FORMAT_DATE(week, "%d %B %Y")
```

- **Refuse half-filled texts.** If a snippet with an open placeholder should never reach the clipboard, end the program before the clipboard line. Put this in front of `IF cmd$ = "copy" AND ALSO:`

```
IF LEN(filled{"missing"}) > 0 AND ALSO cmd$ = "copy" THEN
    PRINT "Give a value for: "; JOIN(filled{"missing"}, ", ")
END 1
```

```
ENDIF
```

When it goes wrong

- **"No snippet called late"**: the file is not in the folder the settings name, or it does not end in `.txt`. `list` shows what the tool sees.
- **"Not a snippet name"**: the name holds a space, a dot or a slash. Use `late_reply` or `late-reply` instead.
- **"A snippet called late exists; use --force"**: `add` keeps the old text. Add `--force` if you really want to replace it.
- **"Use name=value"**: a `--set` without `=`. Quote values that hold spaces: `-s "name=Anna Berg"`.
- **A placeholder stays open although you gave it**: the name differs. `list` shows the names the snippet expects; `{{due_date}}` needs `-s due_date=...`, not `-s due=...`.

Balance dividend

About 20 minutes a week of searching the sent folder, and no more mails that greet Anna as Peter.

M14 Contract Diff

The chore

A supplier sends Jonas the new version of a framework contract. "Only small changes," the mail says. The supplier switched off track changes, and Word's own comparison marks every renumbered clause and every changed format until the page is more red than black. So Jonas prints both versions and reads them side by side with a ruler. Twelve pages, an hour, and the nagging feeling that he missed the one sentence that counted.

What you get

Give the recipe the old and the new version of a Word document. It reads both, compares them paragraph by paragraph and tells you what happened to each paragraph that is not the same in both:

- **changed:** the paragraph is still there, with some words different. The report marks the words: [-100-] was removed, {+120+} was added.
- **added** and **removed:** a paragraph that exists in only one of the versions.
- **moved:** the same text at a different place.

The report is printed and saved as a Word file you can forward, with a heading per change and the text before and after.

Before you start

The setup wizard has run. You need both versions as .docx files; an old .doc or a PDF has to be saved as .docx in Word first. The recipe does not change either file.

The program

The program reads the two file names from the command line and leaves the comparison to the module:

```
' Contract Diff: compares two versions of a Word document paragraph by
' paragraph, prints what changed and writes the same report as a Word
' file. The settings are the [contract_diff] part of work.conf.
```



```

'
'   jdbcbasic contract_diff.jdb "offer v1.docx" "offer v2.docx"
'   jdbcbasic contract_diff.jdb v1.docx v2.docx --dry-run   print only

IMPORT WORKCONF, CONTRACTDIFF, CLI

DIM spec = CLI.NEW("contract_diff", "what changed in a Word document")
CLI.ARG(spec, "before", "the older version", TRUE)
CLI.ARG(spec, "after", "the newer version", TRUE)
CLI.FLAG(spec, "dry-run", "", "print the report, write no file")
CLI.OPT(spec, "config", "", "the settings file", "")
DIM argv = APPEND(["contract_diff.exe"], DROP(1, OS.ARGS()))
DIM got = CLI.PARSE(spec, argv)
IF CLI.DONE(spec, got) THEN END CLI.STATUS(got)

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "contract_diff."
DIM home$ = "~/Documents/AutomateWork/reports"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "reports", home$)
DIM like$ = "" + WORKCONF.VALUE(cfg, key$ + "similar_percent", "60")
DIM percent = VAL(like$)

DIM v1$ = got{"args"}[0]
DIM v2$ = got{"args"}[1]
DIM older = CONTRACTDIFF.PARAGRAPHS(v1$)
DIM newer = CONTRACTDIFF.PARAGRAPHS(v2$)
DIM changes = CONTRACTDIFF.COMPARE(older, newer, percent / 100)
PRINT CONTRACTDIFF.REPORT$(changes)

IF LEN(changes) > 0 AND ALSO NOT got{"dry-run"} THEN
    DIM name1$ = PATH.BASENAME$(v1$)
    DIM name2$ = PATH.BASENAME$(v2$)
    name1$ = LEFT$(name1$, LEN(name1$) - LEN(PATH.EXT$(name1$)))
    name2$ = LEFT$(name2$, LEN(name2$) - LEN(PATH.EXT$(name2$)))
    MKDIR folder$
    DIM out_name$ = "changes " + name1$ + " to " + name2$ + ".docx"
    DIM report$ = PATH.JOIN$(folder$, out_name$)
    CONTRACTDIFF.REPORT_DOCX(changes, name1$, name2$, report$)
    PRINT
    PRINT "Report: "; report$
ENDIF

```

The module reads the paragraphs, pairs them up and writes the report:

```

' contractdiff.jdb: what changed between two versions of a document,
' paragraph by paragraph: the changed ones (with the changed words
' marked), the added, the removed and the moved ones.

```

```

'
'   IMPORT CONTRACTDIFF
'   DIM v1 = CONTRACTDIFF.PARAGRAPHS("offer v1.docx")
'   DIM v2 = CONTRACTDIFF.PARAGRAPHS("offer v2.docx")
'   PRINT CONTRACTDIFF.REPORT$(CONTRACTDIFF.COMPARE(v1, v2))

EXPORT MODULE CONTRACTDIFF

IMPORT DOCX, TEXTDIFF

DIM gLF$ = CHR$(10)

' The paragraphs of a Word file with text in them, trimmed.
EXPORT FUNC PARAGRAPHS(path$)
    IF NOT FILE.EXISTS(path$) THEN THROW "No such document: " + path$
    DIM all = DOCX.PARAGRAPHS(path$)
    DIM cleaned = SELECT(LAMBDA p$ -> _
        TRIM$(REPLACE$(p$, CHR$(13), "")), all)
    RETURN FILTER(LAMBDA p$ -> p$ <> "", cleaned)
ENDFUNC

' How alike two paragraphs are, from 0 to 1, by their words.
EXPORT FUNC LIKENESS(a$, b$)
    RETURN TEXTDIFF.RATIO(TEXTDIFF.TOKENS(a$), TEXTDIFF.TOKENS(b$))
ENDFUNC

' The paragraphs outside the runs both versions share: [number, text]
' pairs of the old version (gone) and the new one (came), from 1.
FUNC Unmatched(before, after)
    DIM gone = []
    DIM came = []
    DIM ops = TEXTDIFF.OPCODES(before, after)
    DIM k = 0
    DIM i = 0
    FOR k = 0 TO LEN(ops) - 1
        DIM op = ops[k]
        IF op{"tag"} <> "equal" THEN
            FOR i = op{"a1"} TO op{"a2"} - 1
                DIM was$ = before[i]
                PUSH(gone, [i + 1, was$])
            NEXT i
            FOR i = op{"b1"} TO op{"b2"} - 1
                DIM arrived$ = after[i]
                PUSH(came, [i + 1, arrived$])
            NEXT i
        ENDIF
    NEXT k
    RETURN [gone, came]

```

```
ENDFUNC
```

```
FUNC Change(kind$, old_no, new_no, before$, after$)
```

```
    DIM c AS MAP
```

```
    c{"kind"} = kind$
```

```
    c{"old_no"} = old_no
```

```
    c{"new_no"} = new_no
```

```
    c{"old"} = before$
```

```
    c{"new"} = after$
```

```
    c{"marked"} = ""
```

```
    IF kind$ = "changed" THEN
```

```
        c{"marked"} = TEXTDIFF.WORDDIFF$(before$, after$)
```

```
    ENDIF
```

```
    RETURN c
```

```
ENDFUNC
```

```
' The changes from the old paragraphs to the new ones, in the order of
' the new version: maps with "kind" (changed, added, removed, moved),
' "old_no" and "new_no" (0 when there is none), "old", "new" and, for
' a changed one, "marked" with [-removed-] and {+added+} words. A pair
' at least as alike as similar (0.6) counts as one changed paragraph.
```

```
EXPORT FUNC COMPARE(before, after, similar = 0.6)
```

```
    DIM parts = Unmatched(before, after)
```

```
    DIM gone = parts[0]
```

```
    DIM came = parts[1]
```

```
    DIM used = ZEROS(LEN(came))
```

```
    DIM out = []
```

```
    DIM g = 0
```

```
    DIM c = 0
```

```
    FOR g = 0 TO LEN(gone) - 1
```

```
        DIM gp = gone[g]
```

```
        DIM g_text$ = gp[1]
```

```
        DIM best = -1
```

```
        DIM best_score = 0.0
```

```
        FOR c = 0 TO LEN(came) - 1
```

```
            IF used[c] = 0 THEN
```

```
                DIM cp = came[c]
```

```
                DIM c_text$ = cp[1]
```

```
                DIM score = LIKENESS(g_text$, c_text$)
```

```
                IF score > best_score THEN
```

```
                    best_score = score
```

```
                    best = c
```

```
            ENDIF
```

```
        ENDIF
```

```
    NEXT c
```

```
    IF best >= 0 AND ALSO best_score >= similar THEN
```

```
        used[best] = 1
```

```
        DIM hit = came[best]
```

```

        DIM hit_text$ = hit[1]
        DIM kind$ = "changed"
        IF hit_text$ = g_text$ THEN kind$ = "moved"
        DIM pair = Change(kind$, gp[0], hit[0], g_text$, _
            hit_text$)
        PUSH(out, pair)
    ELSE
        PUSH(out, Change("removed", gp[0], 0, g_text$, ""))
    ENDIF
NEXT g
FOR c = 0 TO LEN(came) - 1
    IF used[c] = 0 THEN
        DIM left_over = came[c]
        DIM new_text$ = left_over[1]
        PUSH(out, Change("added", 0, left_over[0], "", new_text$))
    ENDIF
NEXT c
RETURN SortChanges(out)
ENDFUNC

' The place of a change in the new version; a removed paragraph sits
' just before the new place of its old number.
FUNC OrderKey(c)
    IF c{"new_no"} > 0 THEN RETURN c{"new_no"}
    RETURN c{"old_no"} - 0.5
ENDFUNC

FUNC SortChanges(changes)
    DIM keys = SELECT(LAMBDA c -> OrderKey(c), changes)
    RETURN SELECT(LAMBDA USE(changes) i -> changes[i], GRADE(keys))
ENDFUNC

' How many changes of each kind there are.
EXPORT FUNC SUMMARY(changes)
    DIM s = {"changed": 0, "added": 0, "removed": 0, "moved": 0}
    DIM k = 0
    FOR k = 0 TO LEN(changes) - 1
        DIM kind$ = changes[k>{"kind"}
        s{kind$} = s{kind$} + 1
    NEXT k
    RETURN s
ENDFUNC

' The title line of one change, such as "Paragraph 4 changed".
EXPORT FUNC TITLE$(c)
    DIM kind$ = c{"kind"}
    DIM was$ = "Paragraph " + STR$(c{"old_no"})
    DIM now_at$ = "Paragraph " + STR$(c{"new_no"})

```

```

    IF kind$ = "added" THEN RETURN now_at$ + " added"
    IF kind$ = "removed" THEN RETURN was$ + " removed"
    DIM moved_to$ = " moved to " + STR$(c{"new_no"})
    IF kind$ = "moved" THEN RETURN was$ + moved_to$
    RETURN now_at$ + " changed"
ENDFUNC

' The report as plain text.
EXPORT FUNC REPORT$(changes)
    DIM s = SUMMARY(changes)
    DIM out = []
    DIM head$ = STR$(s{"changed"}) + " changed, " + STR$(s{"added"})
    head$ = head$ + " added, " + STR$(s{"removed"}) + " removed, "
    PUSH(out, head$ + STR$(s{"moved"}) + " moved")
    DIM k = 0
    FOR k = 0 TO LEN(changes) - 1
        DIM c = changes[k]
        PUSH(out, "")
        PUSH(out, TITLE$(c))
        IF c{"kind"} = "changed" THEN
            PUSH(out, " " + c{"marked"})
        ELSEIF c{"kind"} = "added" THEN
            PUSH(out, " " + c{"new"})
        ELSE
            PUSH(out, " - " + c{"old"})
        ENDIF
    NEXT k
    RETURN JOIN(out, gLF$)
ENDFUNC

' The report as a Word file: a summary, then one heading per change
' with the text before and after.
EXPORT FUNC REPORT_DOCX(changes, old_name$, new_name$, _
    path$)
    DIM doc = DOCX.NEW()
    DOCX.LANGUAGE(doc, "en-US")
    DOCX.TITLE(doc, "Changes " + old_name$ + " to " + new_name$)
    DOCX.HEADING(doc, "What changed")
    DIM from$ = "From *" + old_name$ + "*" to "*" + new_name$ + "*"
    DOCX.PARAGRAPH(doc, from$)
    DIM s = SUMMARY(changes)
    DIM kinds = ["changed", "added", "removed", "moved"]
    DIM counts = SELECT(LAMBDA USE(s) k$ -> STR$(s{k$}), kinds)
    DOCX.TABLE(doc, [kinds, counts])
    DIM k = 0
    FOR k = 0 TO LEN(changes) - 1
        DIM c = changes[k]
        DOCX.HEADING(doc, TITLE$(c), 2)
    
```

```

    IF c{"old"} <> "" AND ALSO c{"kind"} <> "moved" THEN
        DOCX.PARAGRAPH(doc, "***Before:** " + c{"old"})
    ENDIF
    IF c{"new"} <> "" THEN
        DOCX.PARAGRAPH(doc, "***After:** " + c{"new"})
    ENDIF
    IF c{"kind"} = "changed" THEN DOCX.CODE(doc, c{"marked"})
NEXT k
RETURN DOCX.WRITE(doc, path$)
ENDFUNC

```

How it works

1. **From a file to paragraphs.** DOCX.PARAGRAPHS answers the text of every paragraph in the document, without the formatting. PARAGRAPHS trims each one with a SELECT and leaves out the empty ones with a FILTER, so an extra blank line in the new version is no change.
1. **What both versions share.** TEXTDIFF.OPCODES is the same comparison that programmers use for source code, applied to paragraphs instead of lines. It answers a list of steps: this run of paragraphs is equal in both, that run was replaced by another one. Unmatched keeps only what is not equal: the paragraphs that left the old version (gone) and those that came into the new one (came), each with its number.
1. **Pairing the leftovers.** A paragraph with one word changed shows up twice in that list, once as gone and once as came. COMPARE pairs them up again. For every gone paragraph it looks for the came paragraph that is most alike. LIKENESS measures that with TEXTDIFF.RATIO over the words: 1 means the same words in the same order, 0 means nothing in common. If the best match is the same text, the paragraph moved. If it reaches the similar threshold (0.6 unless you change it), the paragraph changed. Otherwise the old paragraph counts as removed, and every came paragraph without a partner counts as added.
1. **Marking the words.** For a changed paragraph, Change asks TEXTDIFF.WORDDIFF\$ for the words that differ. That is the line with [- and {+ you see in the report.
1. **In reading order.** SortChanges puts the changes in the order of the new version, so you can read the report next to the new document. OrderKey

gives each change its place: its paragraph number in the new version, or, for a removed paragraph, a place just before where its old number would be. A SELECT takes the place of every change, GRADE answers their order, and a second SELECT picks the changes in that order.

1. **Two reports from one list.** REPORT\$ writes the text you see on the screen and REPORT_DOCX the Word file. Both read the same list of changes, so they always agree.

Run it

The recipe's test uses a short offer in two versions. Run on those two files, it prints:

```
jdbasic contract_diff.jdb "offer v1.docx" "offer v2.docx"
2 changed, 1 added, 1 removed, 0 moved

Paragraph 3 changed
  The price is [-100-]{+120+} EUR per month.

Paragraph 4 removed
  - Payment within 30 days.

Paragraph 4 changed
  Support by mail {+and phone +}on working days.

Paragraph 5 added
  + A setup fee of 300 EUR applies.

Report: C:\Users\jonas\Documents\AutomateWork\reports\
        changes offer v1 to offer v2.docx
```

The price clause swapped places with the term and got a new price at the same time. The recipe reports it once, as changed, under its number in the new version. The term itself is reported as nothing: its text is the same, and the price moving past it is enough. With `--dry-run` the recipe prints the report and writes no file. When the two versions are the same, it says 0 changed, 0 added, 0 removed, 0 moved and writes nothing either.

Schedule it

Contracts arrive when they arrive, so the wizard plans no task for this recipe. Keep the command at hand instead. In Explorer, select both files, hold Shift, right-click and choose *Copy as path*; paste that after `jdbasic contract_diff.jdb` and the two names are in place.

Make it yours

The settings are in the part of `work.conf` that starts with `[contract_diff]`:

```
[contract_diff]
reports = "~/Documents/AutomateWork/reports"
similar_percent = 60
```

`similar_percent` decides when two paragraphs are one changed paragraph and when they are a removed and an added one. Raise it to 80 for legal texts, where a paragraph with half its words different deserves to be read as new. Lower it to 40 if the report shows too many removed and added pairs that are really the same clause rewritten.

Three changes in the code are worth knowing:

- **Leave out page lines.** Some documents carry "Page 3 of 12" as paragraphs, and every new page in the new version then counts as a change. In PARAGRAPHS, change the FILTER that keeps a paragraph into:

```
RETURN FILTER(LAMBDA p$ -> p$ <> "" ANDALSO _
    NOT STARTSWITH(p$, "Page "), cleaned)
```

- **Send the report to legal.** Put the report into the outbox, by the outbox rule of this chapter. Add OUTBOX, MAIL to the IMPORT line of the program, and these lines after `PRINT "Report: "; report$`:

```
DIM subject$ = "Changes " + name1$ + " to " + name2$
DIM msg = MAIL.MESSAGE(OUTBOX.FROM$(cfg), "legal@example.com", _
    subject$)
MAIL.PLAIN(msg, CONTRACTDIFF.REPORT$(changes))
MAIL.ATTACH(msg, report$)
PRINT "In the outbox: "; OUTBOX.PUT$(msg, cfg, subject$ + ".eml")
```


- **Leave out what only moved.** When the order of the clauses does not matter to you, filter the list right after COMPARE in the program:

```
changes = FILTER(LAMBDA c -> c{"kind"} <> "moved", changes)
```

When it goes wrong

- **"No such document":** a file name with spaces needs quotes around it, as in the example above.
- **Everything is removed and added:** the new version is a scan or was pasted as one long paragraph. Compare the paragraph counts: open both files and look at the pilcrow marks (*Show all* in Word's *Home* ribbon).
- **A table cell change is missing:** the recipe compares paragraphs of the main text. Text inside tables, headers and footers is not part of DOCX.PARAGRAPHS.
- **Many small changes in one long paragraph:** the marked line gets hard to read. Open the Word report; it shows the paragraph before and after in full, below the marked line.

Balance dividend

About 30 minutes a week for a team lead who signs contracts, and a report the lawyer can read without you.

M15 Timesheet for the Boss

The chore

On the first working day of every month, Jonas owes his department head a timesheet: the hours of the last month, week by week and per project, signed. He tracks his time with the Time Tracker of E07, so the numbers exist. Still, he opens the log in Excel, filters the month, builds a pivot table, copies it week by week into the company's Word template, adds up the totals, prints, signs, scans and mails it. That is an hour of work on a document whose content was finished the moment the month ended.

What you get

Once a month the recipe reads the Time Tracker's log and writes last month's timesheet as a Word file:

- a table for every week that has work in it, one row per project and one column per day, with the totals of each day and each project,
- a table with the totals per project for the whole month,
- lines for your signature and your boss's approval.

It also writes the mail that carries the timesheet to your boss and puts it into the outbox, by the outbox rule of this chapter. You look at it, and send it with --send when you are ready.

Before you start

The Time Tracker (E07) has been running for a month. The recipe finds its log on its own: `log.csv` in the folder you gave E07 in the wizard, the folder of the `[time_tracker]` part of `work.conf`. The `[mail]` part holds your mail server, as the outbox rule describes, and the recipe's own part names your boss:

```
[timesheet]
boss = "Anna Berg <anna.berg@example.com>"
```

Without boss the recipe still writes the timesheet, and only the mail is left out.

The program

The program decides which month to write, writes it once and puts the mail into the outbox:

```
' Timesheet for the Boss: turns last month of the Time Tracker's log
' into a Word timesheet and puts the mail that carries it into the
' outbox. The settings are the [timesheet] part of work.conf, the mail
' server the [mail] part.
'
'   jdbasic monthly_timesheet.jdb                last month, once
'   jdbasic monthly_timesheet.jdb --month 2026-10  a given month
'   jdbasic monthly_timesheet.jdb --dry-run         show, write nothing
'   jdbasic monthly_timesheet.jdb --send           send the outbox

IMPORT WORKCONF, TIMESHEET, OUTBOX

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())

IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
    PRINT result{"sent"}; " sent"
    DIM f = 0
    FOR f = 0 TO LEN(result{"failed"}) - 1
        PRINT "  not sent: "; result{"failed"}[f]
    NEXT f
    END
ENDIF

DIM key$ = "timesheet."
DIM e07$ = "~/Documents/AutomateWork/time"
DIM time_dir$ = WORKCONF.FOLDER$(cfg, "time_tracker.folder", e07$)
DIM log$ = WORKCONF.FOLDER$(cfg, key$ + "log", "")
IF log$ = "" THEN log$ = PATH.JOIN$(time_dir$, "log.csv")
DIM sheets_home$ = "~/Documents/AutomateWork/timesheets"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", sheets_home$)
DIM boss$ = WORKCONF.VALUE(cfg, key$ + "boss", "")
DIM person$ = WORKCONF.VALUE(cfg, "user.name", "")
DIM now$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M:%S")
DIM month$ = WORKCONF.ARG$("month")
DIM asked = month$ <> ""
IF NOT asked THEN month$ = TIMESHEET.LAST_MONTH$(LEFT$(now$, 10))

DIM sheet$ = PATH.JOIN$(folder$, "Timesheet " + month$ + ".docx")
IF FILE.EXISTS(sheet$) AND ALSO NOT asked THEN
    PRINT "The timesheet for "; month$; " is done already."
    END
ENDIF
```

```

DIM work = TIMESHEET.SESSIONS(TIMESHEET.EVENTS(log$))
DIM totals = TIMESHEET.MONTH_TOTALS(work, month$, now$)
DIM projects = SORT(MAP.KEYS(totals))
IF LEN(projects) = 0 THEN
    PRINT "No work logged in "; month$
    END
ENDIF

IF WORKCONF.FLAG("dry-run") THEN
    PRINT "Timesheet "; TIMESHEET.MONTH_NAME$(month$); ", "; person$
    DIM p = 0
    FOR p = 0 TO LEN(projects) - 1
        DIM h = totals{projects[p]}
        PRINT " "; projects[p]; ": "; FORMAT$("{:.2f}", h); " hours"
    NEXT p
    PRINT "would write "; sheet$
    IF boss$ <> "" THEN PRINT "would put a mail to "; boss$
    END
ENDIF

MKDIR folder$
DIM hours = TIMESHEET.DOCUMENT(sheet$, person$, month$, work, now$)
PRINT "Wrote "; sheet$; " ("; FORMAT$("{:.2f}", hours); " hours)"
IF boss$ = "" THEN
    PRINT "No [timesheet] boss in work.conf, so no mail."
ELSE
    DIM from$ = OUTBOX.FROM$(cfg)
    DIM msg = TIMESHEET.MESSAGE(from$, boss$, person$, month$, _
        sheet$, hours)
    DIM eml_name$ = "timesheet " + month$ + ".eml"
    PRINT "In the outbox: "; OUTBOX.PUT$(msg, cfg, eml_name$)
    PRINT "Send it with: jdbasic monthly_timesheet.jdb --send"
ENDIF

```

The module turns the log into weeks, tables and the Word file:

```

' timesheet.jdb: a month of the Time Tracker's log (E07) as a
' timesheet for the boss: a Word file with a table per week, the
' totals per project and lines to sign, and the mail that carries it.
'
'   IMPORT TIMESHEET
'   DIM work = TIMESHEET.SESSIONS(TIMESHEET.EVENTS("log.csv"))
'   TIMESHEET.DOCUMENT("sheet.docx", "Jonas", "2026-10", work, now$)

EXPORT MODULE TIMESHEET

```

```

IMPORT DOCX, MAIL

' The events of the Time Tracker's log as rows [time, event, project].
EXPORT FUNC EVENTS(log_path$)
  IF NOT FILE.EXISTS(log_path$) THEN
    THROW "No time log at " + log_path$
  ENDIF
  DIM kinds = ["STRING", "STRING", "STRING"]
  RETURN CSVREADER(log_path$, ",", TRUE, kinds)
ENDFUNC

' The blocks of work: maps with "project", "start" and "stop"
' (" " while it runs). A start while another block runs ends it.
EXPORT FUNC SESSIONS(events)
  DIM out = []
  DIM running = FALSE
  DIM cur = {"project": "", "start": "", "stop": ""}
  DIM k = 0
  FOR k = 0 TO LEN(events) - 1
    DIM ev = events[k]
    DIM when$ = ev[0]
    DIM what$ = ev[1]
    IF running THEN
      cur{"stop"} = when$
      PUSH(out, cur)
      running = FALSE
    ENDIF
    IF what$ = "start" THEN
      DIM project$ = ""
      IF LEN(ev) > 2 THEN project$ = ev[2]
      cur = {"project": project$, "start": when$, "stop": ""}
      running = TRUE
    ENDIF
  NEXT k
  IF running THEN PUSH(out, cur)
  RETURN out
ENDFUNC

' A day (YYYY-MM-DD) at noon, where an hour more or less from
' daylight saving time keeps the date.
FUNC Noon(day$)
  RETURN CDATE(day$ + " 12:00:00")
ENDFUNC

' The month before the one a day (YYYY-MM-DD) is in, as YYYY-MM.
EXPORT FUNC LAST_MONTH$(day$)
  DIM first$ = LEFT$(day$, 8) + "01"
  DIM before = DATEADD("D", -1, Noon(first$))

```

```

    RETURN LEFT$(FORMAT_DATE(before, "%Y-%m-%d"), 7)
ENDFUNC

' The Monday of the week a day (YYYY-MM-DD) belongs to.
EXPORT FUNC MONDAY$(day$)
    DIM back = (VAL(FORMAT_DATE(Noon(day$), "%w")) + 6) MOD 7
    RETURN FORMAT_DATE(DATEADD("D", -back, Noon(day$)), "%Y-%m-%d")
ENDFUNC

FUNC DayAfter$(day$, n)
    RETURN FORMAT_DATE(DATEADD("D", n, Noon(day$)), "%Y-%m-%d")
ENDFUNC

' The Mondays of every week that has a day in the month (YYYY-MM).
EXPORT FUNC WEEKS(month$)
    DIM out = []
    DIM monday$ = MONDAY$(month$ + "-01")
    DO WHILE LEFT$(monday$, 7) <= month$
        PUSH(out, monday$)
        monday$ = DayAfter$(monday$, 7)
    LOOP
    RETURN out
ENDFUNC

' Hours of a block; a running block counts until now$.
FUNC Hours(s, now$)
    DIM stop$ = s{"stop"}
    IF stop$ = "" THEN stop$ = now$
    RETURN DATEDIFF("N", CDATE(s{"start"}), CDATE(stop$)) / 60
ENDFUNC

FUNC Two$(h)
    RETURN FORMAT$("{:.2f}", h)
ENDFUNC

' Hours with two decimals, or an empty cell for none.
FUNC Cell$(h)
    IF h = 0 THEN RETURN ""
    RETURN Two$(h)
ENDFUNC

' Hours per project and day for one week: a map from project to seven
' numbers, Monday first; days outside the month stay 0.
EXPORT FUNC WEEK_HOURS(sessions, monday$, month$, now$)
    DIM table AS MAP
    DIM k = 0
    FOR k = 0 TO LEN(sessions) - 1
        DIM s = sessions[k]

```

```

    DIM day$ = LEFT$(s{"start"}, 10)
    DIM col = DATEDIFF("D", Noon(monday$), Noon(day$))
    DIM inside = LEFT$(day$, 7) = month$
    IF col >= 0 ANDALSO col <= 6 ANDALSO inside THEN
        DIM project$ = s{"project"}
        IF NOT MAP.EXISTS(table, project$) THEN
            table{project$} = ZEROS(7)
        ENDIF
        DIM row = table{project$}
        row[col] = row[col] + Hours(s, now$)
        table{project$} = row
    ENDIF
NEXT k
RETURN table
ENDFUNC

' The table of a week: a header with the dates, a row per project with
' its total, and a row with the totals of each day.
EXPORT FUNC WEEK_ROWS(table, monday$, month$)
    DIM names = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
    DIM head = ["Project"]
    DIM d = 0
    FOR d = 0 TO 6
        DIM date$ = DayAfter$(monday$, d)
        DIM dd$ = MID$(date$, 8, 2) + "." + MID$(date$, 5, 2) + "."
        DIM label$ = names[d] + " " + dd$
        IF LEFT$(date$, 7) <> month$ THEN label$ = names[d]
        PUSH(head, label$)
    NEXT d
    PUSH(head, "Total")
    DIM rows = [head]
    DIM day_sums = ZEROS(7)
    DIM projects = SORT(MAP.KEYS(table))
    DIM p = 0
    FOR p = 0 TO LEN(projects) - 1
        DIM hours = table{projects[p]}
        FOR d = 0 TO 6
            day_sums[d] = day_sums[d] + hours[d]
        NEXT d
        DIM cells = SELECT(LAMBDA h -> Cell$(h), hours)
        PUSH(rows, APPEND([projects[p]], _
            APPEND(cells, [Two$(SUM(hours))])))
    NEXT p
    DIM foot = SELECT(LAMBDA h -> Two$(h), day_sums)
    PUSH(rows, APPEND(["Total"], _
        APPEND(foot, [Two$(SUM(day_sums))])))
    RETURN rows
ENDFUNC

```

```

' Hours per project for the whole month.
EXPORT FUNC MONTH_TOTALS(sessions, month$, now$)
  DIM out AS MAP
  DIM k = 0
  FOR k = 0 TO LEN(sessions) - 1
    DIM s = sessions[k]
    IF LEFT$(s{"start"}, 7) = month$ THEN
      DIM project$ = s{"project"}
      DIM h = 0.0
      IF MAP.EXISTS(out, project$) THEN h = out{project$}
      out{project$} = h + Hours(s, now$)
    ENDIF
  NEXT k
  RETURN out
ENDFUNC

' The hours of all projects together.
FUNC MonthHours(totals)
  RETURN SUM(SELECT(LAMBDA USE(totals) n$ -> totals{n$}, _
    MAP.KEYS(totals)))
ENDFUNC

' The table of the month: a row per project with its hours and a
' row with the total.
EXPORT FUNC TOTAL_ROWS(totals)
  DIM names = SORT(MAP.KEYS(totals))
  DIM body = SELECT(LAMBDA USE(totals) n$ -> _
    [n$, Two$(totals{n$})], names)
  DIM rows = APPEND([["Project", "Hours"]], body)
  DIM foot = ["Total", Two$(MonthHours(totals))]
  PUSH(rows, foot)
  RETURN rows
ENDFUNC

' A month written out, such as "October 2026".
EXPORT FUNC MONTH_NAME$(month$)
  RETURN FORMAT_DATE(CDATE(month$ + "-01"), "%B %Y")
ENDFUNC

' The timesheet as a Word file; answers the hours of the month.
EXPORT FUNC DOCUMENT(path$, person$, month$, sessions, now$)
  DIM doc = DOCX.NEW()
  DOCX.LANGUAGE(doc, "en-US")
  DIM title$ = "Timesheet " + MONTH_NAME$(month$)
  DOCX.TITLE(doc, title$ + ", " + person$)
  DOCX.HEADING(doc, title$)
  DOCX.PARAGRAPH(doc, "***" + person$ + "***")

```



```

DIM mondays = WEEKS(month$)
DIM k = 0
FOR k = 0 TO LEN(mondays) - 1
    DIM monday$ = mondays[k]
    DIM table = WEEK_HOURS(sessions, monday$, month$, now$)
    IF LEN(MAP.KEYS(table)) > 0 THEN
        DOCX.Heading(doc, "Week from " + monday$, 2)
        DOCX.TABLE(doc, WEEK_ROWS(table, monday$, month$), TRUE)
    ENDIF
NEXT k
DIM totals = MONTH_TOTALS(sessions, month$, now$)
DOCX.Heading(doc, "Totals per project", 2)
DOCX.TABLE(doc, TOTAL_ROWS(totals), TRUE)
DOCX.PARAGRAPH(doc, "")
DIM sign$ = " _____ Date: _____"
DOCX.PARAGRAPH(doc, "Employee:" + sign$)
DOCX.PARAGRAPH(doc, "Approved:" + sign$)
DOCX.WRITE(doc, path$)
RETURN MonthHours(totals)
ENDFUNC

' The mail that carries the timesheet to the boss.
EXPORT FUNC MESSAGE(from$, to$, person$, month$, path$, hours)
    DIM subject$ = "Timesheet " + MONTH_NAME$(month$) + ", " + person$
    DIM msg = MAIL.MESSAGE(from$, to$, subject$)
    DIM body$ = "Hello," + CHR$(10) + CHR$(10)
    body$ = body$ + "attached is my timesheet for "
    body$ = body$ + MONTH_NAME$(month$)
    body$ = body$ + ", " + Two$(hours) + " hours in all." + CHR$(10)
    body$ = body$ + CHR$(10) + "Best regards" + CHR$(10) + person$
    MAIL.PLAIN(msg, body$)
    MAIL.ATTACH(msg, path$)
    RETURN msg
ENDFUNC

```

How it works

1. **From the log to blocks of work.** The log holds one line per start or stop. EVENTS reads it with CSVREADER, every column as text, and SESSIONS turns the lines into blocks with a project, a start and a stop. It follows the rules of the Time Tracker: a start while another project runs ends that one, and a block without a stop is still running. Hours counts a running block until now, so a forgotten stop shows up as a long day instead of a missing one.

1. **Which month.** `LAST_MONTH$` takes the first day of the current month and goes back one day; the month of that day is the one before. That works in January as well, where *the month before* is December of the previous year.
1. **Weeks that start in another month.** A month rarely starts on a Monday. `MONDAY$` finds the Monday of the week a day belongs to, with the number `FORMAT_DATE` gives for `%w`: 0 for Sunday, 1 for Monday and so on. `WEEKS` starts at the Monday of the week with the first of the month and steps forward seven days at a time while the Monday is still in the month. October 2026 starts on a Thursday, so its first week begins on 28 September.
1. **Only the days of the month.** `WEEK_HOURS` adds the hours of every block to its project and its day of the week, but only for days in the month. The September days of that first week stay empty, so the work on them is not counted twice, once in September's sheet and once in October's. `WEEK_ROWS` turns the hours into a table: a header with the day and the date (no date for the days outside the month), a row per project, and the totals. A `SELECT` writes the seven cells of a project, empty for a day without hours, and `SUM` adds up its row; the row of totals is a `SELECT` over the sums of the days.
1. **The document.** `DOCUMENT` writes the title, your name, a heading and a table for every week that has work in it, the totals of the month from `TOTAL_ROWS`, which makes a row per project with a `SELECT`, and the two signature lines. It answers the hours of the month, which the mail mentions.
1. **The mail.** `MESSAGE` writes a short text to your boss and attaches the file. The program puts it into the outbox with `OUTBOX.PUT$`. Nothing in this recipe sends anything; `--send` sends what is in the outbox, after it has asked for your password.
1. **Once a month.** The program writes a timesheet only if the file does not exist yet. Run it as often as you like: after the first time in a month it answers that the work is done.

Run it

```
jdbasic monthly_timesheet.jdb --dry-run
Timesheet September 2026, Jonas Weber
Admin: 2.25 hours
Miller offer: 12.00 hours
```

```

Team: 2.00 hours
would write C:\Users\jonas\Documents\AutomateWork\timesheets\
           Timesheet 2026-09.docx
would put a mail to Anna Berg <anna.berg@example.com>

jdbasic monthly_timesheet.jdb
Wrote C:\Users\jonas\Documents\AutomateWork\timesheets\
           Timesheet 2026-09.docx (16.25 hours)
In the outbox: C:\Users\jonas\Documents\AutomateWork\outbox\
           timesheet 2026-09.eml
Send it with: jdbasic monthly_timesheet.jdb --send

jdbasic monthly_timesheet.jdb
The timesheet for 2026-09 is done already.

```

Open the .eml file in the outbox to see the mail as your boss will get it, with the timesheet attached. --month 2026-08 writes the sheet of another month and overwrites a sheet that is there already, which is what you want after you corrected the log.

Schedule it

The wizard plans the recipe for the first of every month at 08:00, with the schedule monthly 1 08:00. Chapter 6 shows how that schedule got into the wizard.

The Task Scheduler does not catch up a task whose time passed while the computer was off, and the first of a month falls on a weekend in about two months out of seven. If your computer is off on those days, set the schedule to weekly mon 08:00 instead. The recipe needs nothing else for that: on the first Monday of a month the sheet of the month before does not exist yet, so it is written; on every other Monday the recipe sees the file and ends.

Sending stays with you. The mail waits in the outbox until you run jdbasic monthly_timesheet.jdb --send yourself.

Make it yours

The settings are in the part of work.conf that starts with [timesheet]:

```

[timesheet]
log = ""
folder = "~/Documents/AutomateWork/timesheets"
boss = "Anna Berg <anna.berg@example.com>"

```

The name on the sheet is your name from the [user] part, and the sender of the mail comes from the [mail] part. An empty log means the Time Tracker's own log, so moving E07 to another folder in the wizard moves M15 with it. Set log only when the log lives somewhere the Time Tracker does not write.

Three changes in the code are worth knowing:

- **Round to quarter hours.** Many companies book time in quarters of an hour. In Hours, replace the line with RETURN DATEDIFF by two lines that round every block up to the next quarter:

```
DIM minutes = DATEDIFF("N", CDATE(s{"start"}), CDATE(stop$))
RETURN CEIL(minutes / 15) / 4
```

- **Leave out private time.** If you track lunch or a doctor's appointment as a project of its own, drop it before anything is counted. In the program, add this line after the one that sets work:

```
work = FILTER(LAMBDA s -> s{"project"} <> "Private", work)
```

- **Your company's words.** The text of the mail is in MESSAGE, the signature lines are at the end of DOCUMENT. Change "Approved:" to "Department head:", or add a line with your personnel number:

```
DOCX.PARAGRAPH(doc, "Personnel number: 4711")
```

When it goes wrong

- **"No time log at":** the Time Tracker has not written a log yet, or you keep it somewhere else. Leave log empty to use the one in E07's folder, or set it to the file; jdbasic time_tracker.jdb status in E07 shows whether the Time Tracker sees its log.
- **"No work logged in 2026-09":** the log has no block that starts in that month. Check that the Time Tracker ran, and the month you gave with --month.
- **A day with 15 hours:** a stop was forgotten, and the block ran on until the next start. Correct the line in log.csv as E07 describes, then write the month again with --month.

- **"No [timesheet] boss in work.conf, so no mail.":** add boss to the settings, then run with `--month` to write the sheet and the mail once more.
- **The mail does not go out:** `--send` reports every message it could not send, and the reason. The message stays in the outbox until the next try.

Balance dividend

About 15 minutes a week, counted over the month: an hour of copying on the first working day, and a timesheet that adds up without anyone checking.

Bonus: M16 Macro Player

The chore

Jonas's team orders its supplies through an old order program. It has no import and no interface for other programs, so every order is typed in by hand: open a new order, enter the supplier, the article and the person who ordered it, save. Twenty orders a week, each one the same dozen clicks and key strokes in the same places.

What you get

You write down the clicks and key strokes once, in a small text file called a macro. The Macro Player plays it: it brings the window to the front, types, presses keys and clicks where you said. It writes a log line per step and stops at the first step that goes wrong, so it never types into the wrong window.

Before you start

The setup wizard has run, and the folder of your macros exists:
~/Documents/AutomateWork/macros unless you chose another one. Copy the sample `new_order.txt` from the recipe's folder into it and change it for your program. To find the points your macro clicks, start `jdbasic` `macro_player.jdb --where`, point at each place in turn, and read its position on the screen.

Watch out

While a macro plays, the Macro Player takes over your keyboard and your mouse. Keep your hands off both until it is done. To stop it at once, hold `Esc`, or push the mouse into the top-left corner of the screen; it checks both before every single step. Never run a macro on a locked screen or one others can see or use, and ask your IT department first: some companies forbid programs that type and click for you.

The program

The program finds the macro you name, checks it and plays it:

```
' Macro Player: plays a macro, a text file of keyboard and mouse
' steps, from your folder of macros. The settings are the
' [macro_player] part of work.conf.
'
'   jdbasic macro_player.jdb new_order           play a macro
'   jdbasic macro_player.jdb new_order --dry-run  show its steps
'   jdbasic macro_player.jdb --list              the macros you have
'   jdbasic macro_player.jdb --where             where the mouse is

IMPORT WORKCONF, INPUTKIT, MACROS

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "macro_player."
DIM home$ = "~/Documents/AutomateWork/macros"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM wait_s = WORKCONF.VALUE(cfg, key$ + "countdown", 5)
DIM log$ = PATH.JOIN$(folder$, "macro_player.log")

IF WORKCONF.FLAG("where") THEN
    PRINT "Point at the places your macro clicks; 10 seconds."
    DIM w = 0
    FOR w = 1 TO 20
        DIM at = INPUTKIT.CURSOR_AT()
        PRINT MACROS.WHERE_LINE$(at[0], at[1])
        SLEEP 500
    NEXT w
END
ENDIF

IF WORKCONF.FLAG("list") THEN
    DIM names = MACROS.LIST(folder$)
    IF LEN(names) = 0 THEN PRINT "No macros in "; folder$
    DIM i = 0
    FOR i = 0 TO LEN(names) - 1
        PRINT names[i]
    NEXT i
END
ENDIF

DIM name$ = MACROS.NAME_ARG$(OS.ARGS())
IF name$ = "" THEN
    PRINT "Name a macro: jdbasic macro_player.jdb new_order"
END 1
ENDIF
```

```

DIM job = MACROS.LOAD(MACROS.PATH$(folder$, name$))
IF LEN(job{"errors"}) > 0 THEN
    PRINT "The macro has mistakes:"
    PRINT " "; JOIN(job{"errors"}, CHR$(10) + " ")
END 1
ENDIF

DIM dry_run = WORKCONF.FLAG("dry-run")
IF NOT dry_run THEN INPUTKIT.COUNTDOWN(wait_s)
DIM r = MACROS.PLAY(job{"steps"}, name$, log$, dry_run)
IF r{"failed"} <> "" THEN
    PRINT "Stopped after "; r{"done"}; " steps: "; r{"failed"}
END 1
ENDIF
IF dry_run THEN
    DIM n = LEN(job{"steps"})
    PRINT n; " steps. Run without --dry-run to play them."
ELSE
    PRINT "Played "; r{"done"}; " steps of "; name$; "."
ENDIF
ENDIF

```

The module knows the folder of macros, the command line and the log:

```

' macros.jdb: a folder of macro files and how to play one. A macro is
' a text file of INPUTKIT commands; playing it writes one log line per
' step and stops at the first step that fails.
'
'   IMPORT MACROS
'   DIM path$ = MACROS.PATH$("C:/Users/jonas/macros", "invoice")
'   DIM job = MACROS.LOAD(path$)
'   DIM r = MACROS.PLAY(job{"steps"}, "invoice", "player.log", TRUE)

EXPORT MODULE MACROS

IMPORT INPUTKIT

' The file of a macro by its name, with or without ".txt". A name with
' a folder in it is refused, so a macro always comes from the folder.
EXPORT FUNC PATH$(folder$, name$)
    DIM n$ = TRIM$(name$)
    IF n$ = "" THEN THROW "MACROS: name a macro"
    DIM sep = INSTR(n$, "/") >= 0 ORELSE INSTR(n$, "\") >= 0
    IF sep ORELSE INSTR(n$, "..") >= 0 THEN
        THROW "MACROS: a macro name has no folder in it: " + n$
    ENDIF
    IF NOT ENDswith(LCASE$(n$), ".txt") THEN n$ = n$ + ".txt"
    RETURN PATH.JOIN$(folder$, n$)

```



```

ENDFUNC

' The names of the macros in a folder, sorted, without ".txt".
EXPORT FUNC LIST(folder$)
    IF NOT FILE.ISDIR(folder$) THEN RETURN []
    DIM files = SORT(DIR$(PATH.JOIN$(folder$, "*.txt")))
    RETURN SELECT(LAMBDA f$ -> LEFT$(f$, LEN(f$) - 4), files)
ENDFUNC

' A macro file read and checked: {"steps": [...], "errors": [...]}.
EXPORT FUNC LOAD(path$)
    IF NOT FILE.EXISTS(path$) THEN THROW "MACROS: no file " + path$
    RETURN INPUTKIT.PARSE(TXTREADER$(path$))
ENDFUNC

' The macro named first on the command line: the first argument after
' the program that is no option and no option's value.
EXPORT FUNC NAME_ARG$(argv)
    DIM k = 1
    DO WHILE k < LEN(argv)
        DIM a$ = argv[k]
        IF a$ = "--config" THEN
            k = k + 2
        ELSEIF STARTSWITH(a$, "--") THEN
            k = k + 1
        ELSE
            RETURN a$
        ENDIF
    LOOP
    RETURN ""
ENDFUNC

' One line of the log.
EXPORT FUNC LOG_LINE$(when$, name$, s, result$)
    DIM what$ = INPUTKIT.PLAN_LINE$(s)
    RETURN when$ + " " + name$ + " " + what$ + " " + result$
ENDFUNC

' The mouse position as a line for --where.
EXPORT FUNC WHERE_LINE$(x, y)
    RETURN "mouse at " + STR$(x) + " " + STR$(y)
ENDFUNC

' Plays the steps, or with dry_run only prints them, and appends a log
' line per step to log_path$. Stops at the first step that fails.
' Answers {"done": steps carried out, "failed": "" or why}.
EXPORT FUNC PLAY(steps, name$, log_path$, dry_run)
    DIM lines = []

```

```

DIM failed$ = ""
DIM done = 0
DIM k = 0
FOR k = 0 TO LEN(steps) - 1
    DIM s = steps[k]
    DIM now$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M:%S")
    IF dry_run THEN
        PRINT "would "; INPUTKIT.PLAN_LINE$(s)
        PUSH(lines, LOG_LINE$(now$, name$, s, "dry run"))
    ELSE
        TRY
            INPUTKIT.RUN_STEP(s)
            PUSH(lines, LOG_LINE$(now$, name$, s, "ok"))
            done = done + 1
            SLEEP 60
        CATCH
            failed$ = ERRMSG$
            DIM why$ = "failed: " + failed$
            PUSH(lines, LOG_LINE$(now$, name$, s, why$))
        ENDTRY
        IF failed$ <> "" THEN EXITFOR
    ENDIF
NEXT k
IF log_path$ <> "" AND ALSO LEN(lines) > 0 THEN
    DIM before$ = ""
    IF FILE.EXISTS(log_path$) THEN before$ = TXTREADER$(log_path$)
    DIM added$ = JOIN(lines, CHR$(10)) + CHR$(10)
    TXTWRITER log_path$, before$ + added$
ENDIF
RETURN {"done": done, "failed": failed$}
ENDFUNC

```

A macro is plain text, one command per line:

```

# Opens a new order in the order system and fills its header.
# Find the points with: jdbasic macro_player.jdb --where
window Orders - Main Menu
key ctrl+n
wait 800
type Miller Ltd
key tab
type Copy paper, 10 packs
key tab
key tab
type Jonas Example
click 640 520

```

```
wait 300
key ctrl+s
```

The shared module `inputkit.jdb` reads it. Its `PARSE` checks every line and answers the steps, or a list of mistakes with their lines:

```
EXPORT FUNC PARSE(text$)
  DIM steps = []
  DIM errors = []
  DIM src = SPLIT(REPLACE$(text$, CHR$(13), ""), gLF$)
  DIM k = 0
  FOR k = 0 TO LEN(src) - 1
    DIM line_no = k + 1
    DIM raw$ = src[k]
    DIM t$ = TRIM$(raw$)
    IF t$ <> "" AND ALSO NOT STARTSWITH(t$, "#") THEN
      DIM cut = INSTR(t$, " ")
      DIM op$ = LCASE$(t$)
      DIM arg$ = ""
      IF cut >= 0 THEN
        op$ = LCASE$(LEFT$(t$, cut))
        arg$ = MID$(t$, cut + 1)
      ENDIF
      DIM where$ = "line " + STR$(line_no) + ": "
      IF op$ = "type" THEN
        DIM typed$ = MID$(TRIM$(raw$), 5)
        IF typed$ = "" THEN
          PUSH(errors, where$ + "type needs a text")
        ELSE
          PUSH(steps, MakeStep(line_no, op$, typed$))
        ENDIF
      ELSEIF op$ = "window" THEN
        IF TRIM$(arg$) = "" THEN
          PUSH(errors, where$ + "window needs a title")
        ELSE
          PUSH(steps, MakeStep(line_no, op$, TRIM$(arg$)))
        ENDIF
      ELSEIF op$ = "key" THEN
        DIM c = COMBO(arg$, FALSE)
        IF c{"error"} <> "" THEN
          PUSH(errors, where$ + c{"error"})
        ELSE
          DIM spec$ = LCASE$(TRIM$(arg$))
          PUSH(steps, MakeStep(line_no, op$, spec$))
        ENDIF
      ELSEIF op$ = "wait" THEN
        DIM ms$ = TRIM$(arg$)
```

```

        IF NOT IsNumber(ms$) ORELSE VAL(ms$) > 60000 THEN
            DIM need$ = "wait needs 0 to 60000 milliseconds"
            PUSH(errors, where$ + need$)
        ELSE
            PUSH(steps, MakeStep(line_no, op$, ms$))
        ENDIF
    ELSEIF INSTR(gPointer$, "|" + op$ + "|") >= 0 THEN
        DIM one$ = TRIM$(REGEX.REPLACE(" +", arg$, " "))
        DIM xy = SPLIT(one$, " ")
        DIM ok = LEN(xy) = 2
        IF ok THEN ok = IsNumber(xy[0])
        IF ok THEN ok = IsNumber(xy[1])
        IF NOT ok THEN
            DIM two$ = " needs two numbers, x and y"
            PUSH(errors, where$ + op$ + two$)
        ELSE
            DIM s = MakeStep(line_no, op$, "")
            s{"x"} = VAL(xy[0])
            s{"y"} = VAL(xy[1])
            PUSH(steps, s)
        ENDIF
    ELSE
        PUSH(errors, where$ + "unknown command '" + op$ + "'")
    ENDIF
ENDIF
NEXT k
RETURN {"steps": steps, "errors": errors}
ENDFUNC

```

How it works

1. NAME_ARG\$ takes the first word on the command line that is no option, and PATH\$ turns it into a file of the macro folder. A name with a folder in it is refused, so a macro always comes from your folder.
2. LOAD reads the file and INPUTKIT.PARSE checks it. A command it does not know, a click without two numbers, a pause of more than a minute or a key that closes a window, such as Alt+F4, is a mistake. With one mistake in the file, nothing is played.
3. COUNTDOWN gives you a few seconds to take your hands off.
4. PLAY carries out the steps one after the other. window brings a window to the front by its exact title, type types the rest of the line, key presses a key or a combination, click clicks at a point of the screen, wait pauses.

5. Every step writes a line to `macro_player.log` in the macro folder. The first step that fails, such as a window that is not open, ends the run, and its line says why.

Run it

Look at the steps first:

```
jdbasic macro_player.jdb new_order --dry-run
would line 3: bring 'Orders - Main Menu' to the front
would line 4: press ctrl+n
would line 5: wait 800 ms
would line 6: type 'Miller Ltd' (10 characters)
would line 7: press tab
would line 8: type 'Copy paper, 10 packs' (20 characters)
would line 9: press tab
would line 10: press tab
would line 11: type 'Jonas Example' (13 characters)
would line 12: click at 640, 520
would line 13: wait 300 ms
would line 14: press ctrl+s
12 steps. Run without --dry-run to play them.
```

Then open the order program and play it for real with `jdbasic macro_player.jdb new_order. --list` shows the macros of your folder. The log tells what happened:

```
2026-10-02 09:35:12 new_order line 13: wait 300 ms dry run
2026-10-02 09:35:12 new_order line 14: press ctrl+s dry run
```

Schedule it

A macro plays when you start it, because it needs the program open and your hands off the keyboard. Give each macro a shortcut on the desktop that runs `jdbasic macro_player.jdb new_order`, or a button in the Work Cockpit (X06).

Make it yours

The settings are the `[macro_player]` part of `work.conf`:

```
[macro_player]
folder = "~/Documents/AutomateWork/macros"
countdown = 5
```

- **A macro per chore.** Write one file per chore, such as `new_order.txt` and `close_month.txt`, and play each by its name.
- **Waits after dialogs.** A program needs a moment to open a dialog. Put a `wait 500` after every step that opens one; the macro is no faster than the program.
- **Keys before clicks.** `key tab` and `key ctrl+n` work wherever the window is. A click needs the window in the same place every time, so prefer keys where the program offers them.
- **Your own data.** The Form Filler (E16) builds a macro from a list of values; the same idea works for a list of orders, one macro per row of a sheet.

When it goes wrong

- **"no window 'Orders - Main Menu'":** the title is not exactly the one in the title bar, or the program is not open. Copy the title letter by letter; a version number in it counts.
- **The macro has mistakes:** the player names each line and stops before it plays anything:

```
jdbasic macro_player.jdb broken --dry-run
The macro has mistakes:
  line 1: unknown command 'clik'
  line 2: 'q' is not ctrl, shift, alt or win
```

- **Clicks land in the wrong place:** the window moved, or the display scaling of Windows changed. Run `--where` again and correct the points, or use keys instead of clicks. The Desktop Robot (X16) counts its points from the window, so they survive a moved window.
- **"Esc is held: stopped.":** you pressed Esc, or a key got stuck. Finish the order by hand; a macro has no undo.

Balance dividend

About 30 minutes a week for twenty orders, typed the same way every time.

Bonus: M17 Slide Builder

The chore

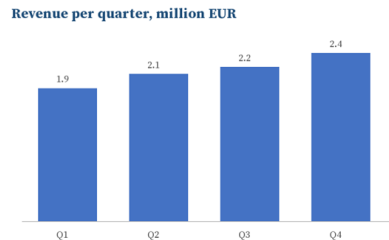
Every quarter Jonas presents the numbers of his team to the head of the department. The content is ready in his notes long before the meeting: a few points per topic, the revenue chart from the report, a table of the regions. What takes the evening is PowerPoint: a new slide, the title into the box, the points into the other box, the chart dragged into place, the table built cell by cell, and the same again for every slide.

What you get

You write the presentation as a short outline in a text file, the way you would write notes, and the Slide Builder turns it into a finished PowerPoint file: a title slide, one slide per topic with its bullet points, pictures and tables placed and sized, and your speaker notes under each slide. If you keep the plan in Excel instead, one row per slide works as well.

Highlights

- Revenue up 9 % on the third quarter
- Two new customers
 - Miller & Sons, three years
 - Berg Logistics, one year
- Delivery time down to two days



A slide the Slide Builder made from three lines of the outline and a chart

Before you start

The setup wizard has run, and you have a folder for your outlines, by default slides in your work folder. The recipe brings a sample, `review.md`, with a chart next to it. You need PowerPoint, or any other program that opens `.pptx` files, only to look at the result; the Slide Builder itself writes the file without it.

An outline is a text file with a few kinds of lines:

```
# Team Review, Fourth Quarter
Jonas Example, Sales team

## Highlights
- Revenue up 9 % on the third quarter
- Two new customers
  - Miller & Sons, three years
  - Berg Logistics, one year
- Delivery time down to two days
![Revenue per quarter](chart.png)
> Thank the team first, then the numbers.

## Revenue by region
| Region | Q3 | Q4 |
|---|---|---|
| North | 1.2 M | 1.3 M |
| South | 0.9 M | 1.1 M |
> South grew most: the new customers are there.

## Next quarter
- Hire one more person for the warehouse
- Move the invoices to the Invoice Generator (M01)
- Review again in January
```

- `#` Title starts the title slide; the plain line below it is its subtitle.
- `##` Title starts a new slide.
- `-` point is a bullet point, and `-` point with two blanks in front one level further in.
- `![text](chart.png)` puts a picture on the slide, from the folder of the outline.
- Lines between `|` signs make a table; the line of dashes under the first row is left out.
- `>` text is a speaker note, which you see in PowerPoint below the slide but the audience does not.

The program

The program finds the outline or the sheet, reads it into slides, shows them, and writes the file:

```
' Slide Builder: turns an outline in Markdown, or a sheet in Excel
' with one row per slide, into a PowerPoint presentation. The
' settings are the [slide_builder] part of work.conf.
,
'   jdbasic slide_builder.jdb review.md           review.pptx
'   jdbasic slide_builder.jdb plan.xlsx           one slide per row
'   jdbasic slide_builder.jdb review.md --dry-run  show the slides

IMPORT WORKCONF, XLSX, SLIDES

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "slide_builder."
DIM home$ = "~/Documents/AutomateWork/slides"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM out_dir$ = WORKCONF.FOLDER$(cfg, key$ + "out_folder", "")
DIM tab$ = WORKCONF.VALUE(cfg, key$ + "tab", "")
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM source$ = SLIDES.SOURCE_ARG$(OS.ARGS())
IF source$ = "" THEN
    PRINT "Name an outline (.md) or a sheet (.xlsx), for example:"
    PRINT "   jdbasic slide_builder.jdb review.md"
    END 1
ENDIF
IF NOT FILE.EXISTS(source$) THEN
    source$ = PATH.JOIN$(folder$, source$)
ENDIF
IF NOT FILE.EXISTS(source$) THEN
    PRINT "No file "; source$
    END 1
ENDIF

DIM base$ = PATH.DIRNAME$(source$)
DIM got AS MAP
IF LCASE$(RIGHT$(source$, 5)) = ".xlsx" THEN
    DIM book = XLSX.READ(source$)
    IF tab$ = "" THEN
        DIM tabs = XLSX.SHEETS(source$)
        tab$ = tabs[0]
    ENDIF
    IF NOT MAP.EXISTS(book, tab$) THEN
        PRINT "The workbook has no sheet "; tab$
        END 1
    ENDIF
ENDIF
```

```

        ENDIF
        got = SLIDES.FROM_ROWS(book{tab$}, base$)
    ELSE
        got = SLIDES.FROM_OUTLINE(TXTREADER$(source$), base$)
    ENDIF

    DIM errors = got{"errors"}
    IF LEN(errors) > 0 THEN
        PRINT "The slides cannot be built:"
        PRINT " "; JOIN(errors, CHR$(10) + " ")
        END 1
    ENDIF
    DIM slides = got{"slides"}
    PRINT JOIN(SLIDES.PLAN_LINES(slides), CHR$(10))
    IF dry_run THEN
        PRINT "Nothing written (dry run)."

```

The module reads both kinds of input into the same list of slides and hands them to the library PPTX, which writes the PowerPoint file:

```

' slides.jdb: the work of the Slide Builder. It reads an outline in
' Markdown or the rows of a sheet into a list of slides, prints them
' for a dry run, and writes them as a PowerPoint file with PPTX.
'
'   IMPORT SLIDES
'   DIM got = SLIDES.FROM_OUTLINE(TXTREADER$("review.md"), ".")
'   PRINT SLIDES.BUILD(got{"slides"}, "Review", "review.pptx")
'
' A slide is a map: "kind" ("title" or "content"), "title",
' "subtitle", "bullets", "image" (a path, "" for none), "table" (rows
' of texts) and "notes".

EXPORT MODULE SLIDES

IMPORT PPTX

FUNC NewSlide(kind$, title$)
    DIM s AS MAP
    s{"kind"} = kind$

```

```

s{"title"} = title$
s{"subtitle"} = ""
s{"bullets"} = []
s{"image"} = ""
s{"table"} = []
s{"notes"} = ""
RETURN s
ENDFUNC

' The cells of a Markdown table line, trimmed.
FUNC Cells(t$)
  DIM inner$ = TRIM$(t$)
  IF LEFT$(inner$, 1) = "|" THEN inner$ = MID$(inner$, 1)
  IF RIGHT$(inner$, 1) = "|" THEN
    inner$ = LEFT$(inner$, LEN(inner$) - 1)
  ENDIF
  RETURN SELECT(LAMBDA c$ -> TRIM$(c$), SPLIT(inner$, "|"))
ENDFUNC

' Whether a table line only separates the header, such as |---|---|.
FUNC IsRule(t$)
  DIM rest$ = REPLACE$(REPLACE$(TRIM$(t$), "|", ""), "-", "")
  rest$ = REPLACE$(rest$, ":", "")
  RETURN TRIM$(rest$) = ""
ENDFUNC

' The slides of an outline: "# Title" starts the title slide (the next
' plain line is its subtitle), "## Title" a slide, "- point" a bullet
' point (" - point" one level in), "![text](file)" a picture, lines
' of "| a | b |" a table, and "> text" the speaker notes. Pictures are
' looked up from base$. Answers "slides" and "errors".
EXPORT FUNC FROM_OUTLINE(text$, base$)
  DIM slides = []
  DIM errors = []
  DIM src = SPLIT(REPLACE$(text$, CHR$(13), ""), CHR$(10))
  DIM cur AS MAP
  DIM have = FALSE
  DIM k = 0
  FOR k = 0 TO LEN(src) - 1
    DIM raw$ = src[k]
    DIM t$ = TRIM$(raw$)
    DIM where$ = "line " + STR$(k + 1) + ": "
    IF t$ = "" THEN CONTINUEFOR
    IF STARTSWITH(t$, "## ") THEN
      IF have THEN PUSH(slides, cur)
      cur = NewSlide("content", TRIM$(MID$(t$, 3)))
      have = TRUE
    ELSEIF STARTSWITH(t$, "# ") THEN

```

```

        IF have THEN PUSH(slides, cur)
        cur = NewSlide("title", TRIM$(MID$(t$, 2)))
        have = TRUE
    ELSEIF NOT have THEN
        PUSH(errors, where$ + "text before the first title")
    ELSEIF STARTSWITH(t$, "- ") ORELSE STARTSWITH(t$, "* ") THEN
        DIM point$ = TRIM$(MID$(t$, 2))
        IF STARTSWITH(raw$, " ") THEN point$ = " " + point$
        DIM points = cur{"bullets"}
        PUSH(points, point$)
        cur{"bullets"} = points
    ELSEIF STARTSWITH(t$, "![") ANDALSO INSTR(t$, "]"") > 0 THEN
        DIM open_at = INSTR(t$, "]"")
        DIM file$ = MID$(t$, open_at + 2, LEN(t$) - open_at - 3)
        DIM pic$ = PATH.JOIN$(base$, file$)
        IF NOT FILE.EXISTS(pic$) THEN
            PUSH(errors, where$ + "no picture " + file$)
        ELSE
            cur{"image"} = pic$
        ENDIF
    ELSEIF STARTSWITH(t$, "|") THEN
        IF NOT IsRule(t$) THEN
            DIM rows = cur{"table"}
            PUSH(rows, Cells(t$))
            cur{"table"} = rows
        ENDIF
    ELSEIF STARTSWITH(t$, ">") THEN
        DIM note$ = TRIM$(MID$(t$, 1))
        DIM before$ = cur{"notes"}
        IF before$ <> "" THEN note$ = before$ + CHR$(10) + note$
        cur{"notes"} = note$
    ELSEIF cur{"kind"} = "title" ANDALSO cur{"subtitle"} = "" THEN
        cur{"subtitle"} = t$
    ELSE
        PUSH(errors, where$ + "not a point, picture, table, note")
    ENDIF
NEXT k
IF have THEN PUSH(slides, cur)
IF LEN(slides) = 0 THEN PUSH(errors, "the outline has no title")
RETURN {"slides": slides, "errors": errors}
ENDFUNC

```

```

' The number of the column named name$ in a header row, -1 if none.
FUNC ColumnOf(header, name$)
    DIM k = 0
    FOR k = 0 TO LEN(header) - 1
        DIM h$ = LCASE$(TRIM$(" " + header[k]))
        IF h$ = name$ THEN RETURN k
    
```

```

    NEXT k
    RETURN -1
ENDFUNC

FUNC Cellof$(row, col)
    IF col < 0 ORELSE col >= LEN(row) THEN RETURN ""
    RETURN TRIM$(" " + row[col])
ENDFUNC

' The slides of a sheet with the columns Title, Points, Picture and
' Notes (Points separated by ";"). The first row under the header is
' the title slide, its Points the line below the title.
EXPORT FUNC FROM_ROWS(rows, base$)
    DIM slides = []
    DIM errors = []
    IF LEN(rows) < 2 THEN
        PUSH(errors, "the sheet needs a header and a row per slide")
        RETURN {"slides": slides, "errors": errors}
    ENDIF
    DIM header = rows[0]
    DIM c_title = ColumnOf(header, "title")
    DIM c_points = ColumnOf(header, "points")
    DIM c_pic = ColumnOf(header, "picture")
    DIM c_notes = ColumnOf(header, "notes")
    IF c_title < 0 THEN
        PUSH(errors, "the sheet has no column Title")
        RETURN {"slides": slides, "errors": errors}
    ENDIF
    DIM r = 0
    FOR r = 1 TO LEN(rows) - 1
        DIM row = rows[r]
        DIM title$ = Cellof$(row, c_title)
        IF title$ = "" THEN CONTINUEFOR
        DIM points$ = Cellof$(row, c_points)
        DIM s AS MAP
        IF LEN(slides) = 0 THEN
            s = NewSlide("title", title$)
            s{"subtitle"} = points$
        ELSE
            s = NewSlide("content", title$)
            DIM parts = SELECT(LAMBDA p$ -> TRIM$(p$), _
                SPLIT(points$, ";"))
            s{"bullets"} = FILTER(LAMBDA p$ -> p$ <> "", parts)
        ENDIF
        DIM file$ = Cellof$(row, c_pic)
        IF file$ <> "" THEN
            DIM pic$ = PATH.JOIN$(base$, file$)
            IF FILE.EXISTS(pic$) THEN

```

```

        s{"image"} = pic$
    ELSE
        DIM row$ = "row " + STR$(r + 1)
        PUSH(errors, row$ + ": no picture " + file$)
    ENDIF
ENDIF
s{"notes"} = CellOf$(row, c_notes)
PUSH(slides, s)
NEXT r
IF LEN(slides) = 0 THEN PUSH(errors, "no row has a title")
RETURN {"slides": slides, "errors": errors}
ENDFUNC

' One line per slide for a dry run.
EXPORT FUNC PLAN_LINES(slides)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(slides) - 1
        DIM s = slides[k]
        DIM line$ = STR$(k + 1) + " " + s{"title"}
        IF s{"kind"} = "title" THEN line$ = line$ + " (title slide)"
        DIM parts = []
        DIM n_points = LEN(s{"bullets"})
        IF n_points > 0 THEN PUSH(parts, STR$(n_points) + " points")
        DIM pic$ = s{"image"}
        DIM pic_name$ = PATH.BASENAME$(pic$)
        IF pic$ <> "" THEN PUSH(parts, "picture " + pic_name$)
        DIM n_rows = LEN(s{"table"})
        IF n_rows > 0 THEN PUSH(parts, STR$(n_rows) + " table rows")
        IF s{"notes"} <> "" THEN PUSH(parts, "notes")
        IF LEN(parts) > 0 THEN
            line$ = line$ + ": " + JOIN(parts, ", ")
        ENDIF
        PUSH(out, line$)
    NEXT k
    RETURN out
ENDFUNC

' Writes the slides as a PowerPoint file; answers how many.
EXPORT FUNC BUILD(slides, title$, path$)
    DIM deck = PPTX.NEW(title$)
    DIM k = 0
    FOR k = 0 TO LEN(slides) - 1
        DIM s = slides[k]
        DIM n = 0
        IF s{"kind"} = "title" THEN
            n = PPTX.TITLE_SLIDE(deck, s{"title"}, s{"subtitle"})
        ELSE

```

```

        n = PPTX.SLIDE(deck, s{"title"}, s{"bullets"})
    ENDIF
    DIM pic$ = s{"image"}
    IF pic$ <> "" THEN PPTX.IMAGE(deck, pic$)
    DIM rows = s{"table"}
    IF LEN(rows) > 0 THEN PPTX.TABLE(deck, rows, TRUE)
    DIM notes$ = s{"notes"}
    IF notes$ <> "" THEN PPTX.NOTES(deck, notes$)
NEXT k
RETURN PPTX.WRITE(deck, path$)
ENDFUNC

' The PowerPoint file for a source file: the same name with .pptx, in
' out_dir$ when it is not "".
EXPORT FUNC TARGET$(source$, out_dir$)
    DIM name$ = PATH.BASENAME$(source$)
    DIM dot = -1
    DIM k = 0
    FOR k = 0 TO LEN(name$) - 1
        IF MID$(name$, k, 1) = "." THEN dot = k
    NEXT k
    IF dot > 0 THEN name$ = LEFT$(name$, dot)
    DIM folder$ = out_dir$
    IF folder$ = "" THEN folder$ = PATH.DIRNAME$(source$)
    IF folder$ = "" THEN RETURN name$ + ".pptx"
    RETURN PATH.JOIN$(folder$, name$ + ".pptx")
ENDFUNC

' A path that does not exist yet: path$ itself, or the same name with
' " (2)", " (3)" and so on before the extension.
EXPORT FUNC FREE_PATH$(path$)
    IF NOT FILE.EXISTS(path$) THEN RETURN path$
    DIM stem$ = LEFT$(path$, LEN(path$) - 5)
    DIM n = 2
    DO WHILE FILE.EXISTS(stem$ + " (" + STR$(n) + ").pptx")
        n = n + 1
    LOOP
    RETURN stem$ + " (" + STR$(n) + ").pptx"
ENDFUNC

' The first argument on the command line that is not an option.
EXPORT FUNC SOURCE_ARG$(argv)
    DIM k = 1
    DO WHILE k < LEN(argv)
        DIM a$ = argv[k]
        IF a$ = "--config" THEN
            k = k + 2
        ELSEIF STARTSWITH(a$, "--") THEN

```

```

        k = k + 1
    ELSE
        RETURN a$
    ENDIF
LOOP
RETURN ""
ENDFUNC

```

How it works

1. `SLIDES.SOURCE_ARG$` takes the first word on the command line that is not an option. If there is no such file, the program looks for it in the folder from `work.conf`.
2. For a `.xlsx` file, `XLSX.READ` reads the workbook and `SLIDES.FROM_ROWS` takes one slide per row with a title. The columns are `Title`, `Points` (separated by `;`), `Picture` and `Notes`; the first row under the header becomes the title slide. A `SELECT` trims the points and a `FILTER` drops the empty ones. Any other file is read as an outline by `SLIDES.FROM_OUTLINE`, where `Cells` trims the cells of a table line with a `SELECT`.
3. Both answer the slides and a list of mistakes, each with its line or row. A picture that is not there, or text before the first title, stops the program before anything is written.
4. `SLIDES.PLAN_LINES` prints one line per slide, which is all a dry run does.
5. `SLIDES.BUILD` creates the deck with `PPTX.NEW` and adds the slides one by one: `PPTX.TITLE_SLIDE` or `PPTX.SLIDE`, then `PPTX.IMAGE`, `PPTX.TABLE` and `PPTX.NOTES` for what the slide holds. `PPTX.WRITE` packs the XML parts of the presentation into the `.pptx` file, which is a ZIP archive like a Word file.
6. `SLIDES.FREE_PATH$` makes sure an older presentation is never overwritten: a second run writes `review (2).pptx`.

Run it

Look at the slides first:

```

jdbasic slide_builder.jdb review.md --dry-run
1 Team Review, Fourth Quarter (title slide)
2 Highlights: 5 points, picture chart.png, notes
3 Revenue by region: 3 table rows, notes

```



```
4 Next quarter: 3 points
Nothing written (dry run).
```

Then without `--dry-run`. The program prints the same lines and the name of the file it wrote, `review.pptx` next to the outline. Open it in PowerPoint: four slides in 16:9, ready to present or to polish.

Schedule it

The Slide Builder has no schedule; a presentation is due when it is due. Start it by hand, or let a recipe that writes a report also write the outline and its chart as a PNG picture, and run the Slide Builder after it.

Make it yours

The settings are the `[slide_builder]` part of `work.conf`:

```
[slide_builder]
folder = "~/Documents/AutomateWork/slides"
out_folder = ""
tab = ""
```

- **One folder for the results.** Set `out_folder` and every presentation lands there instead of next to its outline.
- **A plan in Excel.** Name the sheet in `tab` when the slides are not on the first one.
- **Your colours.** The library draws titles in 1F4E79, a dark blue, and text in 333333. Both are in `lib/pptx.jdb`, in `SlideXml$` and `BulletParas$`; change them there for every deck you build.
- **A slide of your own kind.** Add a line type to `FROM_OUTLINE`, for example `===` quote for a slide with one large sentence, and a `PPTX` call for it in `BUILD`.

When it goes wrong

- **"line 5: no picture chart.png":** the picture is not in the folder of the outline. Copy it there, or write the path relative to the outline.
- **"text before the first title":** the outline starts with a point or a sentence. Put a `#` or `##` title in front.

- **"the sheet has no column Title"**: the first row of the sheet must hold the column names. Add a header row.
- **PowerPoint says the file needs repair**: tell the authors, and send the outline that caused it. Every deck built while this recipe was written opened in PowerPoint without one.

Balance dividend

About 30 minutes a week for a team lead who presents every week or two, and an evening saved before every quarterly review.

Bonus: M18 Excel Refresh

The chore

Every Monday Jonas sends the weekly sales report to his department. The workbook is built: its data comes from the sales database through a connection, pivot tables and formulas do the rest. Still he opens it, clicks *Refresh All*, waits, checks that the numbers moved, saves it, makes a PDF of the summary sheet and writes the mail. Twenty minutes on a good Monday, and on a busy one the report goes out at noon.

What you get

Early on Monday morning Excel Refresh opens the report in Excel, refreshes its connections and pivot tables, recalculates and saves it. Then it makes a PDF of each sheet you chose, with the date in its name, and puts a mail with the PDFs into your outbox. Everything Excel draws stays as it is: formulas, formats, charts and conditional colours, because Excel itself does the work.

Weekly sales by region

Region	Last week	This week	Change
North	41200	44800	+9%
South	38900	36100	-7%
East	27400	29900	+9%
West	33100	35600	+8%
Total	140600	146400	+4%

The summary sheet as the PDF the recipe makes

Before you start

The setup wizard has run, and you have a report workbook that gets its data by itself when you click *Refresh All*: from a database, another workbook, a CSV file or a web page. Write down the names of the sheets that belong in the PDF.

Watch out

The recipe needs Excel installed and licensed on this computer, the desktop version of Microsoft Office. It runs under your own account, so it reaches the data sources with your rights, and it runs from the Task Scheduler while you are logged in, not as a Windows service. A connection that asks for a password each time stops the refresh; store the password in the connection or ask IT for a connection that uses your Windows login.

The program

The program reads the settings, lets the module do the Excel part and puts the mail into the outbox:

```
' Excel Refresh: opens the report workbook in Excel, refreshes its
' data and pivot tables, recalculates and saves it, makes PDFs of the
' chosen sheets, and can put them into the outbox as a mail. The
' settings are the [excel_refresh] part of work.conf.
'
'   jdbasic excel_refresh.jdb           refresh now
'   jdbasic excel_refresh.jdb --dry-run show what it would do
'   jdbasic excel_refresh.jdb --send    send the outbox

IMPORT WORKCONF, REFRESH, OUTBOX

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())

IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
    PRINT result{"sent"}; " sent"
    DIM f = 0
    FOR f = 0 TO LEN(result{"failed"}) - 1
        PRINT "  not sent: "; result{"failed"}[f]
    NEXT f
    END
ENDIF

DIM key$ = "excel_refresh."
DIM book$ = WORKCONF.FOLDER$(cfg, key$ + "workbook", "")
DIM listed = WORKCONF.VALUE(cfg, key$ + "sheets", [])
DIM pdf_home$ = "~/Documents/AutomateWork/reports"
DIM pdf_dir$ = WORKCONF.FOLDER$(cfg, key$ + "pdf_folder", pdf_home$)
DIM mail_to$ = WORKCONF.VALUE(cfg, key$ + "mail_to", "")
DIM day$ = FORMAT_DATE(NOW(), "%Y-%m-%d")
```

```

IF book$ = "" ORELSE NOT FILE.EXISTS(book$) THEN
    PRINT "No workbook at "; book$; "'. "
    PRINT "Set the workbook in the [excel_refresh] part of work.conf."
    END 1
ENDIF
IF WORKCONF.FLAG("dry-run") THEN
    PRINT "would refresh, recalculate and save "; book$
    IF LEN(listed) = 0 THEN
        PRINT "would make a PDF of every sheet in "; pdf_dir$
    ELSE
        PRINT "would make PDFs of "; JOIN(listed, ", ")
        PRINT " in "; pdf_dir$
    ENDIF
    IF mail_to$ <> "" THEN PRINT "would put a mail to "; mail_to$
    END
ENDIF

DIM done = {"pdfs": [], "unknown": []}
TRY
    done = REFRESH.RUN(book$, listed, pdf_dir$, day$)
CATCH
    PRINT "Excel Refresh stopped: "; ERRMSG$
    END 1
ENDTRY

DIM pdfs = done{"pdfs"}
DIM unknown = done{"unknown"}
DIM k = 0
FOR k = 0 TO LEN(unknown) - 1
    PRINT "no sheet named "; unknown[k]
NEXT k
PRINT "Refreshed and saved "; book$
FOR k = 0 TO LEN(pdfs) - 1
    PRINT "made "; pdfs[k]
NEXT k
IF mail_to$ <> "" AND ALSO LEN(pdfs) > 0 THEN
    DIM from$ = OUTBOX.FROM$(cfg)
    DIM msg = REFRESH.MESSAGE(from$, mail_to$, book$, day$, pdfs)
    DIM eml$ = PATH.BASENAME$(book$) + " " + day$ + ".eml"
    PRINT "In the outbox: "; OUTBOX.PUT$(msg, cfg, eml$)
ENDIF

```

The module talks to Excel and writes the mail:

```

' refresh.jdb: the work of Excel Refresh. A report workbook is opened
' in the real Excel, its data connections and pivot tables refreshed,
' recalculated and saved, so its formulas, formats and charts stay;

```

```

' then the chosen sheets become PDFs.
'
'   IMPORT REFRESH
'   DIM book$ = "C:/Reports/sales.xlsx"
'   DIM r = REFRESH.RUN(book$, ["Summary"], "C:/Reports", day$)

EXPORT MODULE REFRESH

IMPORT MAIL

CONST XL_TYPE_PDF = 0

' The PDF of one sheet: the workbook's name, the sheet and the day.
EXPORT FUNC PDF_NAME$(workbook$, sheet$, day$)
    DIM base$ = PATH.BASENAME$(workbook$)
    DIM stem$ = LEFT$(base$, LEN(base$) - LEN(PATH.EXT$(base$)))
    RETURN stem$ + " " + sheet$ + " " + day$ + ".pdf"
ENDFUNC

' The sheets to export, written as the workbook writes them: the
' listed ones in their order, or all sheets when listed is empty.
' Answers {"sheets": [...], "unknown": [names not in the workbook]}.
EXPORT FUNC CHOOSE(listed, available)
    DIM sheets = []
    DIM unknown = []
    IF LEN(listed) = 0 THEN
        RETURN {"sheets": available, "unknown": unknown}
    ENDIF
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(listed) - 1
        DIM want$ = listed[k]
        DIM hit$ = ""
        FOR j = 0 TO LEN(available) - 1
            DIM have$ = available[j]
            IF LCASE$(have$) = LCASE$(want$) THEN hit$ = have$
        NEXT j
        IF hit$ = "" THEN PUSH(unknown, want$) ELSE PUSH(sheets, hit$)
    NEXT k
    RETURN {"sheets": sheets, "unknown": unknown}
ENDFUNC

' Refreshes, recalculates and saves a workbook in Excel, started
' without a window and quit at the end, and writes a PDF of each
' chosen sheet into out_dir$. Answers {"pdfs": [paths], "unknown":
' [sheet names the workbook does not have]}.
EXPORT FUNC RUN(workbook$, listed, out_dir$, day$)
    DIM full$ = REPLACE$(workbook$, "/", "\")

```

```

DIM xl = CREATEOBJECT("Excel.Application")
xl.Visible = FALSE
xl.DisplayAlerts = FALSE
DIM books = xl.Workbooks
DIM wb = books.Open(full$, 0, FALSE)
DIM pdfs = []
DIM unknown = []
TRY
    wb.RefreshAll()
    xl.CalculateUntilAsyncQueriesDone()
    xl.CalculateFull()
    wb.Save()
    DIM wb_sheets = wb.Worksheets
    DIM have = []
    DIM k = 0
    FOR k = 1 TO wb_sheets.Count
        DIM ws = wb_sheets.Item(k)
        DIM ws_name$ = ws.Name
        PUSH(have, ws_name$)
        RELEASEOBJECT(ws)
    NEXT k
    DIM picked = CHOOSE(listed, have)
    unknown = picked{"unknown"}
    DIM names = picked{"sheets"}
    MKDIR out_dir$
    FOR k = 0 TO LEN(names) - 1
        DIM sheet$ = names[k]
        DIM file$ = PDF_NAME$(workbook$, sheet$, day$)
        DIM pdf$ = PATH.JOIN$(out_dir$, file$)
        DIM sh = wb_sheets.Item(sheet$)
        DIM win$ = REPLACE$(pdf$, "/", "\\")
        sh.ExportAsFixedFormat(XL_TYPE_PDF, win$)
        RELEASEOBJECT(sh)
        PUSH(pdfs, pdf$)
    NEXT k
    RELEASEOBJECT(wb_sheets)
CATCH
    DIM why$ = ERRMSG$
    wb.Close(FALSE)
    RELEASEOBJECT(wb)
    RELEASEOBJECT(books)
    xl.Quit()
    RELEASEOBJECT(xl)
    THROW PATH.BASENAME$(workbook$) + ": " + why$
ENDTRY
wb.Close(FALSE)
RELEASEOBJECT(wb)
RELEASEOBJECT(books)

```

```

xl.Quit()
RELEASEOBJECT(xl)
RETURN {"pdfs": pdfs, "unknown": unknown}
ENDFUNC

' The mail that carries the PDFs of a run to the people in to$.
EXPORT FUNC MESSAGE(from$, to$, workbook$, day$, pdfs)
    DIM base$ = PATH.BASENAME$(workbook$)
    DIM subject$ = base$ + ", " + day$
    DIM msg = MAIL.MESSAGE(from$, SPLIT(to$, ","), subject$)
    DIM body$ = "Good morning," + CHR$(10) + CHR$(10)
    body$ = body$ + "the refreshed report " + base$ + " of " + day$
    body$ = body$ + " is attached." + CHR$(10)
    MAIL.PLAIN(msg, body$)
    DIM k = 0
    FOR k = 0 TO LEN(pdf$) - 1
        DIM one$ = pdfs[k]
        MAIL.ATTACH(msg, one$)
    NEXT k
    RETURN msg
ENDFUNC

```

How it works

1. REFRESH.RUN starts Excel with CREATEOBJECT and switches off its window and its questions. It opens the workbook for writing.
2. RefreshAll asks every connection and pivot table for new data, and CalculateUntilAsyncQueriesDone waits until the last one has answered. CalculateFull recalculates every formula, and Save writes the workbook back, so the next person who opens it sees the new numbers.
3. The names of the sheets go through REFRESH.CHOOSE, which finds the ones you listed, no matter how you wrote upper and lower case, and names the ones the workbook does not have.
4. Each chosen sheet becomes a PDF through ExportAsFixedFormat, as weekly sales Summary 2026-10-05.pdf: the workbook, the sheet and the day, so last week's PDF is never overwritten.
5. At the end, and also when something fails, the workbook is closed, Excel quits and every object is let go with RELEASEOBJECT. An error names the workbook and what Excel said.
- 6.

With `mail_to` set, `REFRESH.MESSAGE` builds the mail with the PDFs attached and `OUTBOX.PUT$` puts it into the outbox. It goes out with `--send`, as every mail of the book does.

Run it

See what would happen first:

```
jdbasic excel_refresh.jdb --dry-run
would refresh, recalculate and save C:/Reports/weekly sales.xlsx
would make PDFs of Summary
  in C:/Reports/pdf
would put a mail to sales-team@example.com
```

Then without `--dry-run`. A refresh takes as long as it takes in Excel by hand; the program waits for it and prints the PDFs it made. Open the outbox to read the mail, and send it with `--send`.

Schedule it

The wizard plans it weekly on Monday at seven, before most people read their mail. A report that changes daily can run on weekdays.

Make it yours

The settings are the `[excel_refresh]` part of `work.conf`:

```
[excel_refresh]
workbook = "C:/Reports/weekly sales.xlsx"
sheets = ["Summary"]
pdf_folder = "~/Documents/AutomateWork/reports"
mail_to = "sales-team@example.com"
```

- **Every sheet.** Leave sheets empty, `[]`, and each sheet of the workbook becomes a PDF.
- **No mail.** Leave `mail_to` empty when the PDFs go to a shared folder instead; set `pdf_folder` to that folder.
- **Several reports.** Copy the recipe folder once per workbook, give each copy its own part in `work.conf` by changing `key$` in the program, and plan each at its own time.

When it goes wrong

- **"No workbook at ..."**: the path in workbook is wrong, or the drive is not connected at that hour. Check it in the Explorer.
- **"no sheet named Summery"**: a typing mistake in sheets. The other sheets are still made into PDFs.
- **"Excel Refresh stopped: weekly sales.xlsx: ..."**: Excel could not open, refresh or save the workbook. Most often it is open on another computer and locked, or a data source did not answer. Open it by hand and click *Refresh All* to see Excel's own message.
- **The PDF shows the old numbers**: the connection is set to refresh in the background only when the file opens. In Excel, under *Data, Queries and Connections, Properties*, tick *Refresh data when refreshing all*.

Balance dividend

About 30 minutes a week for anyone who sends a regular report, and a report that is in the mailboxes before the Monday meeting.

Your Own Recipe

Sooner or later the chore that costs you the most time is not in this book. This section takes you from an empty folder to a recipe of your own that the wizard sets up like any other. It starts from a template that does a small job of its own, so you begin with something that works and change it step by step.

The template

The folder `recipes/template/T00_my_recipe` holds a complete recipe: it moves files that nobody changed for a number of days into an archive folder. It has the same five files every recipe in this book has:

File	What it holds
<code>my_recipe.jdb</code>	The program: reads the settings, makes a plan, carries it out
<code>mine.jdb</code>	The module with the work itself, in functions a test can call
<code>my_recipe_test.jdb</code>	The test that proves the module does what it says
<code>recipe.toml</code>	What the wizard needs to know: name, schedule, settings
a page of text	How it works, for you in half a year

The program is short, because it only connects things:

```
' my_recipe.jdb: the template recipe. It moves files nobody touched
' for a while into an archive folder. The settings are the
' [my_recipe] part of work.conf.
'
'   jdbasic my_recipe.jdb           archive now
'   jdbasic my_recipe.jdb --dry-run  only show what would move

IMPORT WORKCONF, MINE

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "my_recipe."
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", "~/Documents")
DIM archive$ = WORKCONF.FOLDER$(cfg, key$ + "archive", _
    PATH.JOIN$(folder$, "Archive"))
DIM days = WORKCONF.VALUE(cfg, key$ + "days", 90)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM moves = MINE.PLAN(folder$, days, NOW())
DIM n = MINE.APPLY(moves, archive$, dry_run)
```

```

IF n = 0 THEN
    PRINT "Nothing older than "; days; " days in "; folder$
ELSEIF dry_run THEN
    PRINT n; " files would move. Run without --dry-run."
ENDIF

```

The work happens in the module, in two functions. PLAN looks and decides; it changes nothing and answers a list. APPLY carries a list out, or prints it on a dry run:

```

' mine.jdb: the work of the template recipe. It plans to move the
' files of a folder that nobody changed for a number of days into an
' archive folder, and carries the plan out. Copy this folder, rename
' it, and replace PLAN and APPLY with the work of your own recipe.
'
'   IMPORT MINE
'   DIM moves = MINE.PLAN("C:/Users/jonas/Reports", 90, NOW())
'   PRINT MINE.APPLY(moves, "C:/Users/jonas/Archive", TRUE)

EXPORT MODULE MINE

' The files of a folder older than days, as maps with "from", "name"
' and "age" in days. Folders inside the folder are left out.
EXPORT FUNC PLAN(folder$, days, now)
    DIM out = []
    DIM names = DIR$(PATH.JOIN$(folder$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM path$ = PATH.JOIN$(folder$, names[k])
        DIM info = FILE.STAT(path$)
        IF NOT info{"is_dir"} THEN
            DIM age = DATEDIFF("D", CDATE(info{"mtime"}), now)
            IF age >= days THEN
                DIM entry = {"from": path$, "name": names[k]}
                entry{"age"} = age
                PUSH(out, entry)
            ENDIF
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' Moves the planned files into archive$, or with dry_run only prints
' them. Answers the number of files.
EXPORT FUNC APPLY(moves, archive$, dry_run)
    DIM k = 0

```

```

FOR k = 0 TO LEN(moves) - 1
    DIM m = moves[k]
    DIM age$ = STR$(INT(m{"age"}))
    IF dry_run THEN
        PRINT "would archive "; m{"name"}; " ("; age$; " days)"
    ELSE
        MKDIR archive$
        FILE.MOVE(m{"from"}, PATH.JOIN$(archive$, m{"name"}))
        PRINT "archived "; m{"name"}
    ENDIF
NEXT k
RETURN LEN(moves)
ENDFUNC

```

Splitting the work this way is the most useful idea of the whole book. A plan can be printed, checked and tested before anything happens on your disk. Every recipe you met in Chapters 3 and 4 works like this.

From template to recipe, step by step

1. **Copy the folder.** Copy T00_my_recipe into recipes/medium and give it a name with a number of your own, such as M50_quote_tracker. Rename the three .jdb files to match.
2. **Describe the chore in one sentence.** Write it as the first line of the program, after the file name. If it does not fit into one sentence, it is two recipes.
3. **Write the plan first.** Change PLAN so that it answers what your recipe would do, as a list of maps, one per action. Print the list from a small test program and look at it. Nothing has happened yet, so you can try as often as you like.
4. **Write the test.** For every case your plan decides, add one check to the test: the old file is in the plan, the new one is not, an empty folder gives an empty plan. Use MKTEMP\$ for a folder of your own, so the test never touches your real files.
5. **Carry the plan out.** Change APPLY. Keep the dry run: print what would happen, and only act when dry_run is false.
6. **Move the settings out.** Every value you might change later, a folder, a number of days, an address, goes into the program's work.conf section through WORKCONF.VALUE or WORKCONF.FOLDER\$, with a sensible default.

7. **Tell the wizard.** Change `recipe.toml`: the id, the name, the program, the module, one `[[setting]]` per setting with its label and default, and a schedule. From now on the wizard lists your recipe with the others.
8. **Run it for a week by hand.** Run it with `--dry-run` every day and compare what it would do with what you would have done. When the two agree for a week, switch the schedule on.

Try this

Before you write a single line, write the "Balance dividend" box of your recipe: how many minutes a week it gives back. If the number is below five, the recipe is probably not worth the time it costs to write. If it is above sixty, check whether it should be two recipes.

The checklist

Run through this list before you schedule a recipe of your own:

- It has a dry run, and the dry run changes nothing.
- It never overwrites a file; a name that is taken gets a number.
- It reads every setting from `work.conf` and has a default for each.
- It needs no password in a file; mail goes through the outbox.
- Its test passes, and the test uses folders of its own.
- `jdbasic --lint` on each of its files finds nothing.
- It prints one line per thing it does, so a log shows what happened.
- Somebody else could read the first comment and know what it is for.

A recipe that passes the list is ready for the Task Scheduler. A recipe that also runs as a compiled program, as every recipe of this book does, is ready to be handed to a colleague; Chapter 5 shows how.

Chapter 5: Expert: Fifteen Programs That Run on Their Own

Lena works in the controlling department of a company with four hundred people. Over two years she has written a dozen small programs for her team: one fetches the bookings every night, one builds the Monday report, one sorts the invoices that arrive by mail. They work. They also break now and then, always on a day she is away, and then somebody calls her because the report did not come.

This chapter is about the step from a program that works when you run it to a program that runs without you. The recipes are larger, between 150 and 400 lines with their modules, and most of them are not single chores but parts of a small system: a server that starts every job on time, checks that tell you when one failed, a place for passwords that is not a text file, a window for colleagues who never open a command prompt.

What changes at the expert level

A program that runs on its own has nobody watching it. Everything a person would notice has to be written down or handled by the program itself. Four habits run through every recipe of this chapter:

1. **Every run leaves a line in a log.** When it started, what it did, how it ended. A log line costs nothing, and it is the only witness you have of a run at three in the morning.
2. **Failures are loud, and only the right ones are retried.** A web server that does not answer may answer in a minute, so the program tries again with a

pause that grows. A missing file or a wrong password will not get better; the program stops and says so, and a health check sends you a mail.

3. **Secrets stay out of files.** API keys, tokens and passwords come from the protected store of Windows, from an environment variable or from a prompt. None of them is written into `work.conf` or into a recipe folder, where a backup or a colleague might copy it.
4. **A test runs before a change goes live.** Each recipe has its test as before. Several of them also test the programs of the earlier chapters, so that a change you make on Thursday does not break the report on Monday.

Talking to other systems

Most recipes of this chapter talk to something: a ticket system, a calendar service, a language model, a web site. Three rules keep that safe for you and polite towards the other side:

- The services the recipes start themselves, such as a search page or a dashboard, listen only on `localhost`. Nobody else on the network can open them.
- A recipe that reads from somebody else's server asks for as little as it needs, waits between requests, and keeps what it fetched in a cache so it does not ask twice.
- Every test talks to a small fake server that the test starts on your computer. No test needs an account, a key or the internet, so you can run all of them on the train.

Where a recipe needs an account with an outside service, the page says which one, what it costs if anything, and what to ask your IT department before you use it at work.

The recipes

Recipe	What it takes off your desk
X01 Personal Job Server	Every automation started on time, with locks, retries and a log

Recipe	What it takes off your desk
X02 Inbox Assistant	Summaries of new mail and reply drafts you approve one by one
X03 Find Anything	A search page over all your documents
X04 Ticket Sync	Tasks kept the same in the ticket system and your to-do list
X05 Calendar Bridge	Your work calendar and your personal one kept in step
X06 Work Cockpit	A window with a button for every automation and its last result
X07 Tools for Colleagues	Recipes as programs your colleagues can run without jdBasic
X08 Parallel Crunching	Thousands of files processed on all cores
X09 Report Pipeline	Database to charts to Word, PDF and mail, every Monday at seven
X10 Let an AI Drive	Your automations offered as tools to an AI assistant
X11 Web Harvester	Data from web pages, collected politely
X12 Tested Automations	Tests that keep a change from breaking Monday's report
X13 Health Checks and Alerts	A mail when a job fails, and numbers for every job
X14 Secrets and Audit Trail	Passwords out of scripts, and a signed record of each run
X15 Balance Score	Your week in one number, and the trend
X16 Desktop Robot (bonus)	An old program driven by keyboard and mouse, checked by screenshots
X17 Meeting Briefing (bonus)	One page per meeting of the day, with mails, open items and a summary
X18 Spreadsheet to Web App (bonus)	An Excel table as a small app in the browser, saved back to Excel
X19 Outlook Bridge (bonus)	Mail sent through Outlook without a password, and Outlook as a source

X01 is the backbone: the other recipes can run under it instead of the Windows Task Scheduler. X13 and X14 make that backbone safe to rely on. Read those three first if you plan to run more than a handful of jobs; the others stand on their own.

X01 Personal Job Server

The chore

Lena's programs started out as entries in the Windows Task Scheduler, one per program, each set up by hand on the day she wrote it. By now there are fourteen of them. When one fails, the Task Scheduler notes a result code that nobody reads. When her laptop is closed at half past six, the evening job does not run, and nothing tells her. Twice the nightly import was still running when the morning report started, and the report showed half a day. Keeping an eye on all of it costs her half an hour a week, and the eye is not always open.

What you get

One program that starts when you log on and runs all your jobs from one file. Each job runs on time, a failed job is tried again with a growing pause, two runs of the same server never happen at once, and a job whose time passed while the computer was off runs once when the server starts. Every run lands in a log and in a state file that other programs read, such as the health check of X13, and with the key of X14 every run is also signed into an audit trail.

```
jdbasic job_server.jdb --list
downloads (daily 18:30)
  next 2026-10-05 18:30, last 2026-10-02 18:30 ok
backup (weekly fri 16:30)
  next 2026-10-09 16:30, last 2026-10-02 16:31 FAILED
status mail (weekly fri 15:00)
  next 2026-10-09 15:00
hot folder (every 5 minutes)
  switched off
```

Before you start

The wizard has set up the recipes the server will run, so they are in the recipes folder of your work folder. Copy the file `jobs.toml` from this recipe's folder into the config folder next to `work.conf`, and write one block per job into it.

Each job the server takes over must stop running on its own. Run the wizard again and set the schedule of those recipes to `manual`; otherwise they run twice, once from the Task Scheduler and once from the server.

The program

The jobs file names each job, when it runs and what it runs. A program is a recipe, given relative to the recipes folder; a command is any other program:

```
# The jobs of the Personal Job Server (X01), one [[job]] per job.
# Copy this file to the config folder of your work folder and edit it;
# the server reads it again when you save it.
#
#   name          a short name of your own
#   when          daily 18:30, weekdays 08:00, weekly fri 16:00,
#                 every 15 minutes, @hourly or a cron line
#   program       a recipe, relative to the recipes folder
#   command       or any other program, with its full path
#   args          what goes after the program
#   attempts      how often to try before it counts as failed (1)
#   retry_seconds the wait before the second try; it doubles (60)
#   catch_up      run once if the computer was off at run time (true)
#   enabled       false switches a job off without deleting it

[[job]]
name = "downloads"
when = "daily 18:30"
program = "E01_downloads_butler/downloads_butler.jdb"

[[job]]
name = "backup"
when = "weekly fri 16:30"
program = "E04_one_click_backup/one_click_backup.jdb"
attempts = 3
retry_seconds = 300

[[job]]
name = "status mail"
when = "weekly fri 15:00"
program = "M06_friday_status/friday_status.jdb"
catch_up = false

[[job]]
name = "hot folder"
when = "every 5 minutes"
program = "M10_hot_folder/hot_folder.jdb"
enabled = false
```

The main program reads the settings and the jobs, and then loops: it runs what is due, writes the state, and sleeps until the next job is due, at most a minute at a time:

```

' Personal Job Server: one program that runs all your automations on
' time. It reads the jobs file, runs each job when it is due, tries a
' failed job again, keeps a log and the state of every job, and signs
' each run into the audit trail when X14 has set up its key. The
' settings are the [job_server] part of work.conf.
'
'   jdbasic job_server.jdb          serve until you log off
'   jdbasic job_server.jdb --list    the jobs and their next runs
'   jdbasic job_server.jdb --dry-run what is due now, run nothing
'   jdbasic job_server.jdb --once    run what is due now, then end
'   jdbasic job_server.jdb --run NAME run one job now
'   jdbasic job_server.jdb --stop    ask a running server to end

IMPORT WORKCONF, JOBSTATE, JOBSERVER, LOGGER, VAULT, DT

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM jobs_path$ = JOBSTATE.JOBS_PATH$(cfg)
DIM state_path$ = JOBSTATE.STATE_PATH$(cfg)
DIM lock$ = JOBSTATE.LOCK_PATH$(cfg)
DIM folder$ = JOBSTATE.FOLDER$(cfg)
DIM stop$ = PATH.JOIN$(folder$, "job_server.stop")
DIM home$ = "~/Documents/AutomateWork"
DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
DIM offset = DT.LOCAL_OFFSET()

IF WORKCONF.FLAG("stop") THEN
    MKDIR folder$
    TXTWRITER stop$, "stop"
    PRINT "The server ends within a minute."
    END
ENDIF

IF NOT FILE.EXISTS(jobs_path$) THEN
    PRINT "No jobs file at "; jobs_path$
    PRINT "Copy jobs.toml of this recipe there and edit it."
    END
ENDIF

DIM jobs = JOBSTATE.JOBS(jobs_path$)
DIM state = JOBSTATE.READ(state_path$)

IF WORKCONF.FLAG("list") THEN
    DIM lines = JOBSERVER.LIST_LINES(jobs, state, DT.NOW(), offset)
    DIM i = 0
    FOR i = 0 TO LEN(lines) - 1
        PRINT lines[i]
    NEXT i
    END
ENDIF

```

```

MKDIR folder$
DIM lg = LOGGER.NEW("jobs")
LOGGER.TO_CONSOLE(lg, FALSE)
LOGGER.TO_FILE(lg, JOBSTATE.LOG_PATH$(cfg), 1000000, 3)
DIM paths AS MAP
paths{"jdbasic"} = WORKCONF.VALUE(cfg, "paths.jdbasic", "jdbasic")
paths{"recipes"} = PATH.JOIN$(work$, "recipes")
paths{"config"} = WORKCONF.CONFIG_PATH$()
DIM ctx AS MAP
ctx{"paths"} = paths
ctx{"log"} = lg
ctx{"offset"} = offset
ctx{"dry_run"} = WORKCONF.FLAG("dry-run")
ctx{"audit"} = PATH.JOIN$(folder$, "audit.log")
ctx{"key"} = VAULT.GET$("AutomateWork/audit-key")

DIM only$ = WORKCONF.ARG$("run")
IF only$ <> "" THEN
    DIM found = FALSE
    DIM j = 0
    FOR j = 0 TO LEN(jobs) - 1
        DIM job = jobs[j]
        IF job{"name"} = only$ THEN
            found = TRUE
            DIM t = DT.NOW()
            DIM res = JOBSERVER.RUN_NOW(job, state, t, t, ctx)
            JOBSTATE.WRITE(state_path$, state)
            IF NOT res{"ok"} THEN PRINT res{"error"}
        ENDIF
    NEXT j
    IF NOT found THEN PRINT "No job named "; only$
END
ENDIF

DIM once = WORKCONF.FLAG("once") ORELSE ctx{"dry_run"}
IF NOT ctx{"dry_run"} THEN
    IF NOT JOBSERVER.FREE(lock$, DT.NOW(), 180) THEN
        PRINT "A job server is running already."
    END
ENDIF
IF FILE.EXISTS(stop$) THEN KILL stop$
LOGGER.INFO(lg, "server started", {"jobs": LEN(jobs)})
ENDIF

DIM stamp$ = FILE.STAT(jobs_path$){ "mtime" }
DO
    DIM now = DT.NOW()

```

```

IF NOT ctx{"dry_run"} THEN JOBSERVER.TOUCH(lock$, now)
DIM results = JOBSERVER.TICK(jobs, state, now, ctx)
IF ctx{"dry_run"} THEN
    IF LEN(results) = 0 THEN PRINT "Nothing is due."
    DIM r = 0
    FOR r = 0 TO LEN(results) - 1
        DIM item = results[r]
        IF MAP.EXISTS(item, "would_run") THEN
            DIM at$ = item{"would_run"}
            PRINT "would run "; item{"name"}; " for "; at$
        ENDIF
    NEXT r
ELSE
    JOBSTATE.WRITE(state_path$, state)
ENDIF
IF once THEN EXITDO
IF FILE.EXISTS(stop$) THEN
    KILL stop$
    EXITDO
ENDIF
DIM changed$ = FILE.STAT(jobs_path$){"mtime"}
IF changed$ <> stamp$ THEN
    stamp$ = changed$
    TRY
        jobs = JOBSTATE.JOBS(jobs_path$)
        LOGGER.INFO(lg, "jobs read again", {"jobs": LEN(jobs)})
    CATCH
        LOGGER.ERROR(lg, "jobs file not used", {"error": ERRMSG$})
    ENDTRY
ENDIF
SLEEP JOBSERVER.NEXT_WAKE(jobs, DT.NOW(), offset) * 1000
LOOP

IF NOT ctx{"dry_run"} THEN
    IF FILE.EXISTS(lock$) THEN KILL lock$
    IF NOT once THEN LOGGER.INFO(lg, "server ended")
ENDIF

```

The module decides what is due and runs it. RUN runs one job with its retries, RUN_NOW records the result, and TICK goes through all jobs once:

```

' jobserver.jdb: the work of the Personal Job Server. It decides which
' jobs are due, runs each one with retries, keeps the state of every
' job and writes the log and the audit trail.
'
'   IMPORT JOBSERVER
'   DIM ctx = {"offset": -7, "dry_run": FALSE, "audit": "", "key": ""}

```

```

'   ctx{"paths"} = {"jdbasic": "jdbasic", "recipes": "C:/W/recipes"}
'   ctx{"log"} = LOGGER.NEW("jobs")
'   DIM results = JOBSERVER.TICK(jobs, state, DT.NOW(), ctx)
'
' A job is due when its last run time before now is later than the run
' time it last ran for. A job the server sees for the first time waits
' for its next run time. A run time that passed with the computer off
' runs once when the server starts, unless catch_up is false.

EXPORT MODULE JOBSERVER

IMPORT SCHED, RETRY, LOGGER, AUDIT, DT, WORKCONF

CONST LATE_SECONDS = 300

' A moment as local time text, "2026-10-02 18:30".
EXPORT FUNC CLOCK$(t, offset)
    RETURN LEFT$(REPLACE$(DT.ISO$(t, offset), "T", " "), 16)
ENDFUNC

' TRUE for a path that starts at a drive or the root.
FUNC Absolute(path$)
    IF STARTSWITH(path$, "/") THEN RETURN TRUE
    IF STARTSWITH(path$, "\\") THEN RETURN TRUE
    RETURN MID$(path$, 1, 1) = ":"
ENDFUNC

' The program and arguments that run a job: a command as it is, a
' program with jdbasic and the settings file of the caller.
EXPORT FUNC COMMAND(job, paths)
    DIM out AS MAP
    IF job{"command"} <> "" THEN
        out{"exe"} = job{"command"}
        out{"args"} = job{"args"}
        RETURN out
    ENDIF
    DIM program$ = WORKCONF.EXPAND$(job{"program"})
    IF NOT Absolute(program$) THEN
        program$ = PATH.JOIN$(paths{"recipes"}, program$)
    ENDIF
    DIM args = APPEND([program$], job{"args"})
    IF MAP.EXISTS(paths, "config") THEN
        DIM conf$ = paths{"config"}
        IF conf$ <> "" THEN args = APPEND(args, ["--config", conf$])
    ENDIF
    out{"exe"} = paths{"jdbasic"}
    out{"args"} = args
    RETURN out

```

```

ENDFUNC

' The last line a program printed.
FUNC LastLine$(output$)
    DIM text$ = TRIM$(REPLACE$(output$, CHR$(13), ""))
    DIM lines = SPLIT(text$, CHR$(10))
    IF LEN(lines) = 0 THEN RETURN ""
    DIM last$ = lines[LEN(lines) - 1]
    RETURN TRIM$(last$)
ENDFUNC

' Runs a command once; an exit code other than 0 is an error.
FUNC Execute$(cmd)
    DIM r = OS.EXEC(cmd{"exe"}, cmd{"args"})
    DIM code = r{"EXIT_CODE"}
    DIM output$ = r{"OUTPUT"}
    IF code = 0 THEN RETURN output$
    DIM last$ = LastLine$(output$)
    IF last$ = "" THEN THROW "exit " + STR$(code)
    THROW "exit " + STR$(code) + ": " + last$
ENDFUNC

' The exit code inside an error text of Execute$, -1 for other errors.
FUNC ExitOf(error$)
    IF NOT STARTSWITH(error$, "exit ") THEN RETURN -1
    DIM colon = INSTR(error$, ":")
    IF colon < 0 THEN RETURN VAL(MID$(error$, 5, LEN(error$)))
    RETURN VAL(MID$(error$, 5, colon - 5))
ENDFUNC

' Runs a job with its retries and answers {"name", "ok", "exit",
' "error", "attempts", "output"}.
EXPORT FUNC RUN(job, paths)
    DIM opts AS MAP
    opts{"attempts"} = job{"attempts"}
    opts{"delay"} = job{"retry_seconds"} * 1000
    opts{"factor"} = 2
    DIM rep = RETRY.ATTEMPT(Execute$@, COMMAND(job, paths), opts)
    DIM out AS MAP
    out{"name"} = job{"name"}
    out{"ok"} = rep{"ok"}
    out{"attempts"} = rep{"attempts"}
    IF rep{"ok"} THEN
        out{"exit"} = 0
        out{"error"} = ""
        out{"output"} = LastLine$(rep{"value"})
    ELSE
        DIM reason$ = rep{"error"}

```



```

        out{"exit"} = ExitOf(reason$)
        out{"error"} = reason$
        out{"output"} = ""
    ENDIF
    RETURN out
ENDFUNC

' The state entry of a job, new when the job has none.
FUNC Entry(state, name$)
    IF MAP.EXISTS(state, name$) THEN RETURN state{name$}
    DIM rec AS MAP
    rec{"runs"} = 0
    rec{"fails"} = 0
    rec{"last_success"} = 0
    RETURN rec
ENDFUNC

' Runs a job now, records the result in its state entry, logs it and
' adds it to the audit trail. Answers the result.
EXPORT FUNC RUN_NOW(job, state, now, due_at, ctx)
    DIM name$ = job{"name"}
    DIM offset = ctx{"offset"}
    DIM rec = Entry(state, name$)
    rec{"last_start"} = CLOCK$(DT.NOW(), offset)
    DIM res = RUN(job, ctx{"paths"})
    DIM finished = DT.NOW()
    rec{"last_end"} = CLOCK$(finished, offset)
    rec{"last_due"} = due_at
    rec{"last_ok"} = res{"ok"}
    rec{"last_exit"} = res{"exit"}
    rec{"last_error"} = res{"error"}
    rec{"runs"} = rec{"runs"} + 1
    DIM fields AS MAP
    fields{"job"} = name$
    fields{"attempts"} = res{"attempts"}
    IF res{"ok"} THEN
        rec{"last_success"} = finished
        LOGGER.INFO(ctx{"log"}, "job ran", fields)
    ELSE
        rec{"fails"} = rec{"fails"} + 1
        fields{"error"} = res{"error"}
        LOGGER.ERROR(ctx{"log"}, "job failed", fields)
    ENDIF
    rec{"next"} = SCHED.NEXTRUN(job{"when"}, now, offset)
    state{name$} = rec
    IF ctx{"key"} <> "" AND ALSO ctx{"audit"} <> "" THEN
        DIM entry AS MAP
        entry{"job"} = name$

```

```

    entry{"ok"} = res{"ok"}
    entry{"exit"} = res{"exit"}
    entry{"attempts"} = res{"attempts"}
    entry{"due"} = CLOCK$(due_at, offset)
    DIM seq = AUDIT.APPEND(ctx{"audit"}, ctx{"key"}, entry)
  ENDF
  RETURN res
ENDFUNC

' What one job does at now: a result when it ran, was skipped or would
' run, an empty map otherwise.
FUNC TickOne(job, state, now, ctx)
  DIM offset = ctx{"offset"}
  DIM name$ = job{"name"}
  DIM due_at = SCHED.PREVRUN(job{"when"}, now + 1, offset)
  DIM rec = Entry(state, name$)
  DIM item AS MAP
  DIM nothing AS MAP
  item{"name"} = name$
  DIM seen = MAP.EXISTS(rec, "last_due")
  IF seen THEN
    IF due_at <= rec{"last_due"} THEN RETURN nothing
    IF ctx{"dry_run"} THEN
      item{"would_run"} = CLOCK$(due_at, offset)
      RETURN item
    ENDIF
    DIM late = now - due_at > LATE_SECONDS
    IF job{"catch_up"} ORELSE NOT late THEN
      RETURN RUN_NOW(job, state, now, due_at, ctx)
    ENDIF
    item{"skipped"} = TRUE
    LOGGER.WARN(ctx{"log"}, "missed run skipped", item)
  ELSE
    LOGGER.INFO(ctx{"log"}, "job added", item)
  ENDIF
  rec{"last_due"} = due_at
  rec{"next"} = SCHED.NEXTRUN(job{"when"}, now, offset)
  state{name$} = rec
  IF seen THEN RETURN item
  RETURN nothing
ENDFUNC

' Runs every enabled job that is due at now and answers one result per
' job it ran, skipped or, on a dry run, would run.
EXPORT FUNC TICK(jobs, state, now, ctx)
  DIM results = []
  DIM k = 0
  FOR k = 0 TO LEN(jobs) - 1

```

```

        DIM job = jobs[k]
        IF job{"enabled"} THEN
            DIM res = TickOne(job, state, now, ctx)
            IF MAP.EXISTS(res, "name") THEN PUSH(results, res)
        ENDIF
    NEXT k
    RETURN results
ENDFUNC

' Seconds until the next job is due, at least 5 and at most 60, so the
' server marks the lock at least once a minute.
EXPORT FUNC NEXT_WAKE(jobs, now, offset)
    DIM active = FILTER(LAMBDA job -> job{"enabled"}, jobs)
    DIM gaps = SELECT(LAMBDA USE(now, offset) job -> _
        SCHED.NEXTRUN(job{"when"}, now, offset) - now, active)
    DIM wait = MIN(APPEND([60], gaps))
    IF wait < 5 THEN wait = 5
    RETURN wait
ENDFUNC

' TRUE when no other server holds the lock: no lock file, or one older
' than ttl seconds.
EXPORT FUNC FREE(lock_path$, now, ttl)
    IF NOT FILE.EXISTS(lock_path$) THEN RETURN TRUE
    DIM t = VAL(TRIM$(TXTREADER$(lock_path$)))
    RETURN now - t > ttl
ENDFUNC

' Marks the lock file with now.
EXPORT SUB TOUCH(lock_path$, now)
    MKDIR PATH.DIRNAME$(lock_path$)
    TXTWRITER lock_path$, STR$(INT(now))
ENDSUB

' Two lines per job for --list: the name with its schedule, then the
' next run and the last result.
EXPORT FUNC LIST_LINES(jobs, state, now, offset)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(jobs) - 1
        DIM job = jobs[k]
        DIM name$ = job{"name"}
        PUSH(out, name$ + " (" + job{"when"} + ")")
        DIM line$ = "  switched off"
        IF job{"enabled"} THEN
            DIM next_t = SCHED.NEXTRUN(job{"when"}, now, offset)
            line$ = "  next " + CLOCK$(next_t, offset)
            IF MAP.EXISTS(state, name$) THEN

```

```

        DIM rec = state{name$}
        IF MAP.EXISTS(rec, "last_ok") THEN
            DIM result$ = "ok"
            IF NOT rec{"last_ok"} THEN result$ = "FAILED"
            DIM ended$ = rec{"last_end"}
            line$ = line$ + ", last " + ended$ + " " + result$
        ENDIF
    ENDIF
ENDIF
PUSH(out, line$)
NEXT k
RETURN out
ENDFUNC

```

Reading the jobs file and the state file is in a small module of its own, `jobstate.jdb` in `recipes/lib`, because the health check and the cockpit read the same files.

How it works

1. **The jobs file.** `JOBSTATE.JOBS` reads `jobs.toml` and fills in every field a job leaves out: one attempt, a minute before the second try, `catch_up` and enabled on. A file with mistakes, such as a name used twice or a schedule `SCHED` cannot read, is an error that lists every mistake at once, so you fix them in one go.
2. **When a job is due.** For each job, `TickOne` asks `SCHED` for the last run time before now. If that time is later than the run time the job last ran for, the job is due. The schedules are the same words the wizard uses: `daily 18:30`, `weekdays 08:00`, `weekly fri 16:00`, `every 15 minutes`, and a cron line such as `30 8 1 * *` for half past eight on the first of each month.
3. **A new job waits.** A job the server sees for the first time gets its last run time as if it had run, and waits for the next one. Adding a job at ten in the morning does not start every daily job at once.
4. **Missed runs.** When the server starts in the morning, a job that was due at half past six last evening is due now: its last run time is later than the time it last ran for. With `catch_up = false` a run that is more than five minutes late is skipped and logged as skipped. That suits the status mail, which nobody wants on Saturday.
5. **Retries.** `RUN` hands the job to `RETRY.ATTEMPT`. The function `Execute$` runs the program with `OS.EXEC` and turns an exit code other than 0 into an

error with the last line the program printed. `RETRY` catches it, waits `retry_seconds`, doubles the wait for the next attempt and gives up after attempts.

6. **The record.** `RUN_NOW` writes start, end, result, exit code and error into the job's entry of the state, counts runs and failures, logs one line and, when `X14` has made its signing key, adds an entry to the audit trail.
7. **One server at a time.** The server writes the time into a lock file each time it wakes, at least once a minute. A second server finds a lock younger than three minutes and ends with *A job server is running already*. The health check reads the same time to see whether the server is alive.
8. **Changes while it runs.** After each round the server looks at the time `jobs.toml` was saved. A new time means it reads the file again; a file with mistakes is logged and the old jobs stay in use. `--stop` leaves a stop file that the server finds within a minute.

Jobs run one after another. A backup that takes twenty minutes holds up the job after it by twenty minutes, which is fine for office work. A program that never ends holds up all of them, so give your own recipes a clear end.

Run it

Look at the plan first. `--list` shows every job with its next run and its last result, and `--dry-run` names what is due right now without running it:

```
jdbasic job_server.jdb --dry-run
would run downloads for 2026-10-05 18:30
```

`--run` runs one job at once, which is the way to try a new job:

```
jdbasic job_server.jdb --run backup
2026-10-05 10:12:40 INFO [jobs] job ran job=backup attempts=1
```

Started without a switch, the server keeps running and prints its log lines:

```
jdbasic job_server.jdb
2026-10-05 08:31:02 INFO [jobs] server started jobs=4
2026-10-05 08:31:02 INFO [jobs] job ran job=downloads attempts=1
```

A job that fails after all its attempts writes an `ERROR` line with *job failed*, the number of attempts and the error.

The log is `job_server.log` in the `logs` folder of your work folder, next to `job_state.json`. When it reaches a megabyte it becomes `job_server.log.1`; the server keeps three such older files and drops the oldest.

Schedule it

The wizard plans the server *at logon*: the Task Scheduler starts it when you log on to Windows, and it runs until you log off. That is the one entry the Task Scheduler keeps; everything else is in `jobs.toml`.

Make it yours

The settings are in the part of `work.conf` that starts with `[job_server]`:

```
[job_server]
jobs = "~/Documents/AutomateWork/config/jobs.toml"
folder = "~/Documents/AutomateWork/logs"
```

- **Once a month.** The wizard can plan a recipe for one day a month (Chapter 6 shows the change), but the Task Scheduler skips a day the computer was off. The server runs a missed job once when it starts. The timesheet of M15 at half past eight on the first of each month:

```
[[job]]
name = "timesheet"
when = "30 8 1 * *"
program = "M15_timesheet/monthly_timesheet.jdb"
```

- **Any program.** A command runs anything with its full path, for example a mirror of a folder to a network drive:

```
[[job]]
name = "mirror"
when = "weekdays 12:30"
command = "C:/Windows/System32/robocopy.exe"
args = ["C:/Reports", "R:/Reports", "/MIR"]
```

robocopy answers 1 when it copied files, which the server would count as a failure. For such a program, run a small batch file instead, `mirror.bat`, that turns every code below 8 into 0:

```
robocopy C:Reports R:Reports /MIR
if %ERRORLEVEL% LSS 8 exit /b 0
exit /b %ERRORLEVEL%
```

and give the job command = "C:/Work/mirror.bat" and no args.

- **What a job said.** To keep the last line every job printed in the log, add one line to the success branch of `RUN_NOW`:

```
fields{"said"} = res{"output"}
```

- **Longer pauses.** A web service that is down is often down for longer than a minute. `attempts = 4` with `retry_seconds = 300` tries again after 5, 10 and 20 minutes. The other jobs wait during those pauses, so keep long pauses for jobs that run at night.

When it goes wrong

- **"A job server is running already", with none running:** the server ended without removing its lock, for example when Windows shut down. The lock counts for three minutes; after that a new server starts. You may also delete `job_server.lock` in the logs folder.
- **"The jobs file has mistakes":** the message lists each one with the name of the job. A running server keeps its old jobs and writes *jobs file not used* into the log until the file is right.
- **A job fails with "exit 1":** run it alone with `--run NAME`. The error has the last line the program printed; run the recipe itself with `--dry-run` to see more.
- **A job runs twice:** its own entry in the Task Scheduler is still there. Set the recipe to `manual` in the wizard.

Balance dividend

About 30 minutes a week of checking whether things ran, plus the mornings that start with a report that is complete.

X02 Inbox Assistant

The chore

Lena starts every day with forty new mails. Most of them need no more than a glance: a newsletter, a copy of something she already knows, a meeting that moved. Six or seven need an answer, and those she writes the same way every time: thank the sender, say what she will do, say when. It takes her the first hour of the morning, and by the time she is done, the first questions of the day are already waiting.

What you get

A language model reads the mails you saved, writes one sentence about each and drafts a reply where the sender expects one. Every draft is a plain text file in a folder of drafts. You read it, change what you want, and approve it by changing one word. Only approved drafts go into the outbox, and only `--send` sends them, with the password you type.

The model never sends anything, and nothing reaches the outbox without you. Where the model would have to promise a date, an amount or a decision, it writes `[CHECK]` instead, and a draft with `[CHECK]` in it stays in the folder until you have filled it in.

Before you start

You need three things:

1. **Saved mails.** The recipe reads `.eml` files from a folder, the same way the Inbox Unpacker (M07) does. Save the mails you want help with into that folder, or let a mail rule do it.
2. **A language model.** Either a service such as OpenAI or Anthropic, or a model on your own computer. For a service you need an API key; put it into the environment variable `OPENAI_API_KEY` (or `ANTHROPIC_API_KEY`), or let the recipe ask for it each time. The key is never written into `work.conf`. For a model of your own, install a model server such as Ollama and set `base_url` to its address; then no key is needed and no mail leaves your computer.

3. **The outbox rule of Chapter 4.** The [mail] part of work.conf names your mail server, as for the other recipes that send mail.

Watch out

A mail you hand to a service goes to that service. Before you use a service for work mail, ask whether your company allows it. Mails with personal data or business secrets are a reason to use a model on your own computer.

The program

The program has four modes: drafting, a dry run, approving and sending. The work is in the module:

```
' Inbox Assistant: a language model reads the mails you saved, writes
' a one line summary of each and drafts a reply. You read every draft,
' change it, approve it, and only then it goes into the outbox. The
' settings are the [inbox_assistant] part of work.conf.
'
' jdbasic inbox_assistant.jdb          draft replies to new mails
' jdbasic inbox_assistant.jdb --dry-run list the mails, ask nothing
' jdbasic inbox_assistant.jdb --approve approved drafts to outbox
' jdbasic inbox_assistant.jdb --send   send the outbox

IMPORT WORKCONF, INBOXAI, MAIL, OUTBOX, LOGGER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "inbox_assistant."
DIM home$ = "~/Documents/AutomateWork"
DIM saved$ = home$ + "/saved mail"
DIM inbox$ = WORKCONF.FOLDER$(cfg, key$ + "inbox", saved$)
DIM own$ = home$ + "/drafts"
DIM drafts$ = WORKCONF.FOLDER$(cfg, key$ + "drafts", own$)
DIM most = WORKCONF.VALUE(cfg, key$ + "max_mails", 20)
DIM me$ = WORKCONF.VALUE(cfg, "user.name", "")
DIM from$ = OUTBOX.FROM$(cfg)
DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
DIM log_dir$ = PATH.JOIN$(work$, "logs")
MKDIR log_dir$
DIM lg = LOGGER.NEW("inbox_assistant")
LOGGER.TO_FILE(lg, PATH.JOIN$(log_dir$, "inbox_assistant.log"))
MKDIR drafts$

IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
```

```

    PRINT "sent "; result{"sent"}; " mail(s)"
END
ENDIF

IF WORKCONF.FLAG("approve") THEN
    DIM ready = INBOXAI.APPROVED(drafts$)
    DIM done$ = PATH.JOIN$(drafts$, "done")
    MKDIR done$
    DIM a = 0
    FOR a = 0 TO LEN(ready) - 1
        DIM name$ = ready[a]
        DIM path$ = PATH.JOIN$(drafts$, name$)
        DIM msg = INBOXAI.TO_MESSAGE(INBOXAI.READ_DRAFT(path$), from$)
        DIM eml$ = REPLACE$(name$, ".txt", ".eml")
        PRINT "outbox "; OUTBOX.PUT$(msg, cfg, eml$)
        FILE.MOVE(path$, done$, TRUE)
        LOGGER.INFO(lg, "approved " + name$)
    NEXT a
    PRINT LEN(ready); " approved draft(s) in the outbox"
END
ENDIF

DIM pending = INBOXAI.PENDING(inbox$, INBOXAI.SEEN(drafts$))
IF LEN(pending) > most THEN pending = TAKE(most, pending)
IF WORKCONF.FLAG("dry-run") THEN
    DIM p = 0
    FOR p = 0 TO LEN(pending) - 1
        PRINT "would ask about "; pending[p]
    NEXT p
END
ENDIF

DIM api_key$ = ""
DIM own_server$ = WORKCONF.VALUE(cfg, key$ + "base_url", "")
IF own_server$ = "" AND ALSO GETENV$("OPENAI_API_KEY") = "" AND ALSO _
    GETENV$("ANTHROPIC_API_KEY") = "" THEN
    INPUT "API key (not stored): "; api_key$
ENDIF
DIM ai = INBOXAI.CLIENT(cfg, api_key$)
DIM drafted = 0
DIM k = 0
FOR k = 0 TO LEN(pending) - 1
    DIM file$ = pending[k]
    TRY
        DIM parsed = MAIL.PARSEFILE(PATH.JOIN$(inbox$, file$))
        DIM d = INBOXAI.DRAFT(ai, parsed, me$)
        PRINT file$; ": "; d{"summary"}
        IF d{"needs_reply"} THEN

```

```

        DIM text$ = INBOXAI.DRAFT_TEXT$(parsed, d, file$)
        DIM draft$ = REPLACE$(file$, ".eml", ".txt")
        TXTWRITER PATH.JOIN$(drafts$, draft$), text$
        drafted = drafted + 1
    ENDIF
    INBOXAI.MARK_SEEN(drafts$, file$)
    DIM about AS MAP
    about{"reply"} = d{"needs_reply"}
    about{"summary"} = d{"summary"}
    LOGGER.INFO(lg, "read " + file$, about)
CATCH
    PRINT file$; ": not read, "; ERRMSG$
    LOGGER.ERROR(lg, "failed " + file$, {"error": ERRMSG$})
ENDTRY
NEXT k
PRINT drafted; " draft(s) in "; drafts$

```

The module talks to the model, writes and reads the draft files, and turns an approved draft into a mail:

```

' inboxai.jdb: the work of the Inbox Assistant. It finds the saved
' mails that have no draft yet, asks a language model for a summary
' and a reply, writes each reply as a draft file for the person to
' read and approve, and turns approved drafts into outbox mails.
'
'   IMPORT INBOXAI
'   DIM ai = INBOXAI.CLIENT(cfg, "")
'   DIM d = INBOXAI.DRAFT(ai, MAIL.PARSEFILE("mail.eml"), "Lena")
'   PRINT d{"summary"}

EXPORT MODULE INBOXAI

IMPORT LLMAPI, RETRY, MAIL, WORKCONF

DIM gLF$ = CHR$(10)

' The language model client from the [inbox_assistant] settings. An
' empty key$ leaves the key to OPENAI_API_KEY or ANTHROPIC_API_KEY.
EXPORT FUNC CLIENT(cfg, key$)
    DIM k$ = "inbox_assistant."
    DIM provider$ = WORKCONF.VALUE(cfg, k$ + "provider", "openai")
    DIM model$ = WORKCONF.VALUE(cfg, k$ + "model", "gpt-4o-mini")
    DIM base$ = WORKCONF.VALUE(cfg, k$ + "base_url", "")
    DIM ai = LLMAPI.NEW(provider$, model$, key$, base$)
    LLMAPI.SET(ai, "temperature", 0.2)
    RETURN ai
ENDFUNC

```

```

' The instructions the model gets with every mail.
EXPORT FUNC RULES$(me$)
    DIM r$ = "You help " + me$ + " with mail. For the mail you get, "
    r$ = r$ + "answer with a JSON object: summary is one sentence "
    r$ = r$ + "about what the sender wants; needs_reply is true "
    r$ = r$ + "when the sender expects an answer; reply is a short "
    r$ = r$ + "polite answer in the language of the mail, signed "
    r$ = r$ + me$ + ", or empty when no answer is needed. Never "
    r$ = r$ + "promise dates, money or decisions: write [CHECK] "
    r$ = r$ + "where " + me$ + " has to fill something in."
    RETURN r$
ENDFUNC

' The mail as the text the model reads, cut to max_chars.
EXPORT FUNC PROMPT$(parsed, max_chars)
    DIM p$ = "From: " + parsed{"from"} + gLF$
    p$ = p$ + "Subject: " + parsed{"subject"} + gLF$
    p$ = p$ + "Date: " + parsed{"date"} + gLF$ + gLF$
    RETURN p$ + LEFT$(TRIM$(parsed{"text"}), max_chars)
ENDFUNC

' A summary and a reply for one parsed mail, as a map with summary,
' needs_reply and reply. A busy or unreachable server is asked again.
EXPORT FUNC DRAFT(ai, parsed, me$)
    LLMAPI.SYSTEM(ai, RULES$(me$))
    DIM shape AS MAP
    shape{"summary"} = "string!"
    shape{"needs_reply"} = "boolean!"
    shape{"reply"} = "string!"
    DIM prompt$ = PROMPT$(parsed, 6000)
    DIM answer AS MAP
    DIM why$ = ""
    DIM got = FALSE
    DIM tries = 0
    FOR tries = 1 TO 3
        TRY
            answer = LLMAPI.JSON(ai, prompt$, shape)
            got = TRUE
        CATCH
            why$ = ERRMSG$
            IF NOT RETRY.TRANSIENT(why$) THEN THROW why$
        ENDTRY
    IF got THEN EXITFOR
    IF tries < 3 THEN SLEEP 500 * tries
NEXT tries
IF NOT got THEN THROW why$
RETURN answer

```

```

ENDFUNC

' The saved mails of inbox$ not yet in the seen list, sorted by name.
EXPORT FUNC PENDING(inbox$, seen)
    DIM names = SORT(DIR$(PATH.JOIN$(inbox$, "*.eml")))
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM name$ = names[k]
        IF NOT Listed(name$, seen) THEN PUSH(out, name$)
    NEXT k
    RETURN out
ENDFUNC

FUNC Listed(name$, names)
    DIM j = 0
    FOR j = 0 TO LEN(names) - 1
        DIM each$ = names[j]
        IF each$ = name$ THEN RETURN TRUE
    NEXT j
    RETURN FALSE
ENDFUNC

' The names in the seen file of the drafts folder, one per line.
EXPORT FUNC SEEN(drafts$)
    DIM path$ = PATH.JOIN$(drafts$, "seen.txt")
    IF NOT FILE.EXISTS(path$) THEN RETURN []
    DIM rows = SPLIT(REPLACE$(TXTREADER$(path$), CHR$(13), ""), gLF$)
    DIM names = SELECT(LAMBDA s$ -> TRIM$(s$), rows)
    RETURN FILTER(LAMBDA s$ -> s$ <> "", names)
ENDFUNC

' Adds a mail's file name to the seen file.
EXPORT SUB MARK_SEEN(drafts$, name$)
    DIM path$ = PATH.JOIN$(drafts$, "seen.txt")
    DIM old$ = ""
    IF FILE.EXISTS(path$) THEN old$ = TXTREADER$(path$)
    TXTWRITER path$, old$ + name$ + gLF$
ENDSUB

' The text of a draft file: header lines, a line ---, the reply.
EXPORT FUNC DRAFT_TEXT$(parsed, d, source$)
    DIM subject$ = parsed{"subject"}
    IF NOT STARTSWITH(LCASE$(subject$), "re:") THEN
        subject$ = "Re: " + subject$
    ENDIF
    DIM t$ = "Approve: no" + gLF$
    t$ = t$ + "To: " + parsed{"from"} + gLF$

```

```

    t$ = t$ + "Subject: " + subject$ + gLF$
    t$ = t$ + "In-Reply-To: " + parsed{"message_id"} + gLF$
    t$ = t$ + "Summary: " + d{"summary"} + gLF$
    t$ = t$ + "Source: " + source$ + gLF$
    RETURN t$ + "----" + gLF$ + TRIM$(d{"reply"}) + gLF$
ENDFUNC

' A draft file read back as a map: approve, to, subject, in_reply_to,
' summary, source and body.
EXPORT FUNC READ_DRAFT(path$)
    DIM text$ = REPLACE$(TXTREADER$(path$), CHR$(13), "")
    DIM cut = INSTR(text$, gLF$ + "----" + gLF$)
    IF cut < 0 THEN THROW "No line --- in " + path$
    DIM d AS MAP
    d{"body"} = TRIM$(MID$(text$, cut + 5))
    DIM rows = SPLIT(LEFT$(text$, cut), gLF$)
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM row$ = rows[k]
        DIM colon = INSTR(row$, ":")
        IF colon > 0 THEN
            DIM key$ = LCASE$(TRIM$(LEFT$(row$, colon)))
            key$ = REPLACE$(key$, "-", "_")
            d{key$} = TRIM$(MID$(row$, colon + 1))
        ENDIF
    NEXT k
    DIM yes$ = ""
    IF MAP.EXISTS(d, "approve") THEN yes$ = LCASE$(d{"approve"})
    d{"approve"} = (yes$ = "yes")
    RETURN d
ENDFUNC

' The drafts the person approved and filled in, as file names. A
' draft that still holds [CHECK] waits for the person.
EXPORT FUNC APPROVED(drafts$)
    DIM names = SORT(DIR$(PATH.JOIN$(drafts$, "*.txt")))
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM name$ = names[k]
        IF name$ <> "seen.txt" THEN
            DIM d = READ_DRAFT(PATH.JOIN$(drafts$, name$))
            DIM open_spot = INSTR(d{"body"}, "[CHECK]") >= 0
            DIM ready = d{"approve"} AND ALSO NOT open_spot
            IF ready THEN PUSH(out, name$)
        ENDIF
    NEXT k
    RETURN out

```

```

ENDFUNC

' An approved draft as a mail, answering the mail it came from.
EXPORT FUNC TO_MESSAGE(d, from$)
  DIM msg = MAIL.MESSAGE(from$, d{"to"}, d{"subject"})
  IF d{"in_reply_to"} <> "" THEN
    MAIL.HEADER(msg, "In-Reply-To", d{"in_reply_to"})
    MAIL.HEADER(msg, "References", d{"in_reply_to"})
  ENDIF
  MAIL.PLAIN(msg, d{"body"})
  RETURN msg
ENDFUNC

```

How it works

1. **The client.** CLIENT builds a client of the LLMAPI library from the settings. The library speaks to OpenAI, to Anthropic and to any server that answers like OpenAI's chat service, which is what Ollama and most local model servers do. The temperature of 0.2 keeps the answers sober and close to the mail.
2. **The rules.** RULES\$ is the instruction the model gets with every mail. It asks for an answer in a fixed shape and forbids promises. This is the text to change when the drafts do not sound like you.
3. **A reply in a fixed shape.** DRAFT hands the mail to LLMAPI.JSON with a shape of three fields: summary, needs_reply and reply. The model answers in exactly that shape, so the program can read the answer without guessing.
4. **Asking again.** A model service is sometimes busy. When the error names a timeout or a busy server, DRAFT waits half a second, then a second, and asks again, three times in all. A wrong key or a broken request is not worth a second try, so RETRY.TRANSIENT decides which errors count as passing.
5. **Each mail once.** PENDING lists the saved mails that are not in the file seen.txt of the drafts folder, and MARK_SEEN adds a mail there after the model read it. A mail the model could not read stays out of seen.txt and comes up again on the next run.
6. **The draft file.** DRAFT_TEXT\$ writes a few header lines, a line with --- and the reply. The first header line is Approve: no. READ_DRAFT reads the file back, and APPROVED lists the drafts that say Approve: yes and hold no [CHECK].

7. **From draft to mail.** `TO_MESSAGE` builds the mail with the headers In-Reply-To and References, so your answer shows up in the sender's mail program in the same thread as the question.
8. **The log.** Every mail read, every failure and every approval is a line in `logs/inbox_assistant.log` of your work folder.

Run it

Start with a dry run. It lists the mails the model would get and asks nothing:

```
jdbasic inbox_assistant.jdb --dry-run
would ask about budget.eml
would ask about news.eml
```

Then let it draft:

```
jdbasic inbox_assistant.jdb
budget.eml: Anna asks for the Q3 figures by Friday.
news.eml: A newsletter with spreadsheet tips.
1 draft(s) in C:\Users\Lena\Documents\AutomateWork\drafts
```

The newsletter needs no reply, so it only gets its sentence. The draft for Anna is the file `budget.txt`:

```
Approve: no
To: Anna Berg <anna@example.com>
Subject: Re: Budget figures
In-Reply-To: <q3@example.com>
Summary: Anna asks for the Q3 figures by Friday.
Source: budget.eml
---
Dear Anna,
you will have them by [CHECK].
Lena
```

Open it in Notepad, write the day instead of `[CHECK]`, change no to yes and save. Then:

```
jdbasic inbox_assistant.jdb --approve
outbox C:\Users\Lena\Documents\AutomateWork\outbox\budget.eml
1 approved draft(s) in the outbox
jdbasic inbox_assistant.jdb --send
```


The approved draft moves to drafts\done, the mail waits in the outbox, and --send asks for the mail password and sends it.

Schedule it

The wizard plans this recipe as *manual*, because a run without a key in the environment stops and asks for one. Once the key is in OPENAI_API_KEY, or once you use a model of your own, set the schedule to weekdays 08:00 in the wizard. The drafts are then waiting when you sit down, and approving them is still your step.

Make it yours

The settings are in the part of work.conf that starts with [inbox_assistant]:

```
[inbox_assistant]
inbox = "~/Documents/AutomateWork/saved mail"
drafts = "~/Documents/AutomateWork/drafts"
provider = "openai"
model = "gpt-4o-mini"
base_url = ""
max_mails = 20
```

- **A model on your own computer.** With Ollama running, set base_url = "http://localhost:11434/v1" and model to the name of a model you pulled, such as "llama3.1". The provider stays "openai", because Ollama answers in OpenAI's shape.
- **Anthropic instead of OpenAI.** Set provider = "anthropic" and a Claude model name, and put the key into ANTHROPIC_API_KEY.
- **Your tone.** Add a sentence to RULES\$, for example r\$ = r\$ + " Write in German when the mail is German." or a line about how formal you are with customers.
- **Fewer mails per run.** max_mails caps how many mails one run hands to the model. A service charges per word, so a low cap keeps a full inbox after a holiday from costing much.
- **Keep the summaries.** The summary of every mail is printed and also written into the log line of that mail, so logs/inbox_assistant.log holds a short history of your inbox. Point the Log Detective (M09) at the logs folder to see when the model failed.

When it goes wrong

- **"answered 401"**: the key is wrong or has expired. The recipe does not ask twice for the same mail; fix the key and run it again.
- **"transport failure"**: the service or your model server cannot be reached. For Ollama, check that it runs and that `base_url` ends in `/v1`. The mail stays pending and comes up on the next run.
- **"did not answer with JSON"**: a small local model could not keep to the shape. Try a larger model, or set the temperature lower in `CLIENT`.
- **An approved draft stays in the folder**: it still holds `[CHECK]` somewhere. Search for it, fill it in, and approve again.

Balance dividend

About 60 minutes a week: the morning hour shrinks to reading the summaries and correcting a few drafts. Keep the time it frees for the mails that need your own thinking.

X03 Find Anything

The chore

Lena knows the document exists. Somebody wrote down last year how the travel costs of the field staff are booked, in a Word file or in the minutes of a meeting, or perhaps in a PDF from the auditors. The shared drive has eleven thousand files in folders named after people who have left. Windows search finds the files whose names contain the word, not the ones that mention it on page four. So she asks around, and twenty minutes later a colleague sends the link.

What you get

A search over the text of all your documents, on a page in your browser: Word files with their tables, PDF files, text, Markdown and CSV. Type a few words and the page lists the best matches, each with the file name, a piece of text around the words it found, and the full path. A misspelt word still finds what you meant.

The index lives in a folder of its own. Each run of the program brings it up to date and reads only the files that are new or changed since the last run, so after the first run an update of a large folder takes seconds.

Before you start

Decide which folders to search, such as your documents and the department's share. Reading a large share the first time takes a while: a few minutes for a thousand Word files. Run the first update by hand and watch it.

The search page, like every server in this chapter, listens only on your own computer.

The program

The program reads the settings and does one of three things: it updates the index, answers a search in the console, or serves the search page:

```

' Find Anything: one search over the documents of your folders, in
' the console or on a page in your browser. Each run brings the index
' up to date and reads only the files that are new or changed. The
' settings are the [find_anything] part of work.conf.
,
'   jdbasic find_anything.jdb                update the index
'   jdbasic find_anything.jdb --dry-run       show what would be read
'   jdbasic find_anything.jdb --search budget search in the console
'   jdbasic find_anything.jdb --serve        the search page

IMPORT WORKCONF, FINDER, JDWEB, LOGGER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "find_anything."
DIM home$ = "~/Documents/AutomateWork/find"
DIM index_dir$ = WORKCONF.FOLDER$(cfg, key$ + "index", home$)
DIM wanted = WORKCONF.VALUE(cfg, key$ + "folders", ["~/Documents"])
DIM kinds = WORKCONF.VALUE(cfg, key$ + "kinds", _
    ["txt", "md", "csv", "docx", "pdf"])
DIM lang$ = WORKCONF.VALUE(cfg, key$ + "language", "en")
DIM port = WORKCONF.VALUE(cfg, key$ + "port", 8766)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM folders = []
DIM k = 0
FOR k = 0 TO LEN(wanted) - 1
    DIM folder$ = WORKCONF.EXPAND$(wanted[k])
    IF NOT FILE.ISDIR(folder$) THEN PRINT "not a folder: "; folder$
    PUSH(folders, folder$)
NEXT k

DIM lg = LOGGER.NEW("find_anything")
LOGGER.TO_CONSOLE(lg, FALSE)
IF NOT dry_run THEN
    MKDIR index_dir$
    LOGGER.TO_FILE(lg, PATH.JOIN$(index_dir$, "find.log"), 1000000, 2)
ENDIF
DIM state = FINDER.OPEN(index_dir$, lang$)

DIM query$ = WORKCONF.ARG$("search")
IF query$ <> "" THEN
    DIM hits = FINDER.FIND(state, query$, 10)
    IF LEN(hits) = 0 THEN PRINT "Nothing found for "; query$
    FOR k = 0 TO LEN(hits) - 1
        DIM hit = hits[k]
        PRINT hit{"path"}
        PRINT "    "; hit{"snippet"}
    NEXT k

```

```

    END
ENDIF

IF WORKCONF.FLAG("serve") THEN
    FINDER.MOUNT(state)
    IF NOT JDWEB.LISTEN(port) THEN
        DIM busy$ = "Port " + STR$(port) + " is taken."
        THROW busy$ + " Set port in [find_anything]."
    ENDIF
    PRINT "Search page: http://localhost:"; port; "/"
    PRINT "Only this computer can open it. Stop it with Ctrl+C."
    HTTP.SERVER.WAIT
    END
ENDIF

DIM plan = FINDER.PLAN(state, FINDER.SCAN(folders, kinds))
DIM news = plan{"add"}
DIM changed = plan{"change"}
DIM gone = plan{"remove"}
IF dry_run THEN
    PRINT LEN(news); " new, "; LEN(changed); " changed, "; _
        LEN(gone); " gone"
    FOR k = 0 TO LEN(news) - 1
        DIM f = news[k]
        PRINT "would read "; f{"path"}
    NEXT k
    FOR k = 0 TO LEN(changed) - 1
        DIM g = changed[k]
        PRINT "would read again "; g{"path"}
    NEXT k
    FOR k = 0 TO LEN(gone) - 1
        PRINT "would drop "; gone[k]
    NEXT k
    END
ENDIF

DIM done = FINDER.UPDATE(state, plan)
FINDER.SAVE(state)
DIM failed = done{"failed"}
FOR k = 0 TO LEN(failed) - 1
    LOGGER.WARN(lg, "not readable: " + failed[k])
NEXT k
LOGGER.INFO(lg, "index up to date", {"read": done{"read"}, _
    "dropped": done{"removed"}, "failed": LEN(failed)})

```

The module does the work:

```

' finder.jdb: the work of Find Anything. It lists the documents of a
' few folders, reads their text, keeps a search index up to date by
' the time each file last changed, and answers a search with a short
' piece of text around the words it found.
'
'   IMPORT FINDER
'   DIM state = FINDER.OPEN("C:/Users/lena/AutomateWork/find", "en")
'   DIM files = FINDER.SCAN(["C:/Users/lena/Reports"], ["docx"])
'   DIM done = FINDER.UPDATE(state, FINDER.PLAN(state, files))
'   FINDER.SAVE(state)
'   DIM hits = FINDER.FIND(state, "travel policy", 10)

EXPORT MODULE FINDER

IMPORT SEARCH, DOCX, HTMLDOM, JDWEB

DIM gState AS MAP

' Every file under folder$ whose extension is one of kinds, as maps
' with "path" (with forward slashes), "mtime" and "size". Hidden files
' and the lock files Office leaves next to an open document are left
' out.
SUB Walk(folder$, kinds, out)
    DIM names = DIR$(PATH.JOIN$(folder$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM name$ = names[k]
        DIM full$ = PATH.JOIN$(folder$, name$)
        DIM info = FILE.STAT(full$)
        IF NOT info{"hidden"} AND ALSO LEFT$(name$, 2) <> "~$" THEN
            IF info{"is_dir"} THEN
                Walk(full$, kinds, out)
            ELSE
                DIM ext$ = LCASE$(MID$(PATH.EXT$(name$), 1, 10))
                IF ext$ IN kinds THEN
                    DIM f AS MAP
                    f{"path"} = REPLACE$(full$, "\", "/")
                    f{"mtime"} = info{"mtime"}
                    f{"size"} = info{"size"}
                    PUSH(out, f)
                ENDIF
            ENDIF
        ENDIF
    NEXT k
ENDSUB

' The documents of all folders; a missing folder adds none.
EXPORT FUNC SCAN(folders, kinds)

```

```

    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(folders) - 1
        DIM folder$ = folders[k]
        IF FILE.ISDIR(folder$) THEN Walk(folder$, kinds, out)
    NEXT k
    RETURN out
ENDFUNC

' The text of one document: Word paragraphs and table cells, the text
' of a PDF, or the file itself for plain text.
EXPORT FUNC TEXT$(path$)
    DIM ext$ = LCASE$(PATH.EXT$(path$))
    IF ext$ = ".pdf" THEN RETURN PDF.TEXT$(path$)
    IF ext$ <> ".docx" THEN RETURN TXTREADER$(path$)
    DIM got = DOCX.READ(path$)
    DIM parts = got{"paragraphs"}
    DIM tables = got{"tables"}
    DIM t = 0
    DIM r = 0
    FOR t = 0 TO LEN(tables) - 1
        DIM rows = tables[t]
        FOR r = 0 TO LEN(rows) - 1
            DIM cells = rows[r]
            PUSH(parts, JOIN(cells, " "))
        NEXT r
    NEXT t
    RETURN JOIN(parts, CHR$(10))
ENDFUNC

' The words of a file name, so "travel_policy-2026.docx" is found as
' travel, policy and 2026.
FUNC NameWords$(path$)
    DIM base$ = PATH.BASENAME$(path$)
    base$ = LEFT$(base$, LEN(base$) - LEN(PATH.EXT$(base$)))
    RETURN REGEX.REPLACE("[_.\-]+", base$, " ")
ENDFUNC

' The index and its catalog from dir$, or new empty ones. lang$ is the
' language for word endings: en, de or none.
EXPORT FUNC OPEN(dir$, lang$)
    DIM state AS MAP
    state{"dir"} = dir$
    DIM ix_path$ = PATH.JOIN$(dir$, "index.txt")
    DIM cat_path$ = PATH.JOIN$(dir$, "catalog.json")
    IF FILE.EXISTS(ix_path$) AND ALSO FILE.EXISTS(cat_path$) THEN
        state{"ix"} = SEARCH.READFILE(ix_path$)
        state{"catalog"} = JSON.PARSE$(TXTREADER$(cat_path$))
    
```

```

ELSE
    DIM opts = {"lang": lang$, "fields": {"name": 2, "body": 1}}
    state{"ix"} = SEARCH.INDEX(opts)
    DIM empty AS MAP
    state{"catalog"} = empty
ENDIF
RETURN state
ENDFUNC

' What an update has to do: the files to read ("add" and "change") and
' the paths to drop from the index ("remove").
EXPORT FUNC PLAN(state, files)
    DIM catalog = state{"catalog"}
    DIM seen AS MAP
    DIM add = []
    DIM change = []
    DIM k = 0
    FOR k = 0 TO LEN(files) - 1
        DIM f = files[k]
        DIM p$ = f{"path"}
        seen{p$} = TRUE
        IF NOT MAP.EXISTS(catalog, p$) THEN
            PUSH(add, f)
        ELSE
            DIM had = catalog{p$}
            DIM old_t$ = had{"mtime"}
            DIM new_t$ = f{"mtime"}
            DIM old_n = had{"size"}
            DIM new_n = f{"size"}
            DIM moved = old_t$ <> new_t$ OR ELSE old_n <> new_n
            IF moved THEN PUSH(change, f)
        ENDIF
    NEXT k
    DIM known = MAP.KEYS(catalog)
    DIM gone = FILTER(LAMBDA USE(seen) old$ -> _
        NOT MAP.EXISTS(seen, old$), known)
    RETURN {"add": add, "change": change, "remove": gone}
ENDFUNC

' Reads one file into the index and the catalog. A file that cannot be
' read is kept in the catalog with its error and is tried again
' after it changes. Answers "" or the error.
FUNC ReadOne$(state, f)
    DIM catalog = state{"catalog"}
    DIM p$ = f{"path"}
    DIM entry AS MAP
    entry{"mtime"} = f{"mtime"}
    entry{"size"} = f{"size"}

```



```

entry{"text"} = ""
DIM problem$ = ""
TRY
    DIM text$ = TEXT$(p$)
    DIM fields = {"name": NameWords$(p$), "body": text$}
    SEARCH.ADD(state{"ix"}, p$, fields)
    entry{"text"} = LEFT$(text$, 4000)
CATCH
    problem$ = ERRMSG$
    DIM ignored = SEARCH.REMOVE(state{"ix"}, p$)
ENDTRY
entry{"error"} = problem$
catalog{p$} = entry
state{"catalog"} = catalog
RETURN problem$
ENDFUNC

' Carries a plan out. Answers "read", "removed" and "failed", the last
' a list of "path: reason" lines.
EXPORT FUNC UPDATE(state, plan)
    DIM failed = []
    DIM n = 0
    DIM k = 0
    DIM lists = [plan{"add"}, plan{"change"}]
    DIM j = 0
    FOR j = 0 TO 1
        DIM todo = lists[j]
        FOR k = 0 TO LEN(todo) - 1
            DIM f = todo[k]
            DIM problem$ = ReadOne$(state, f)
            IF problem$ = "" THEN
                n = n + 1
            ELSE
                PUSH(failed, f{"path"} + ": " + problem$)
            ENDIF
        NEXT k
    NEXT j
    DIM catalog = state{"catalog"}
    DIM gone = plan{"remove"}
    FOR k = 0 TO LEN(gone) - 1
        DIM old$ = gone[k]
        DIM ignored = SEARCH.REMOVE(state{"ix"}, old$)
        MAP.DELETE(catalog, old$)
    NEXT k
    state{"catalog"} = catalog
    RETURN {"read": n, "removed": LEN(gone), "failed": failed}
ENDFUNC

```

```

' Writes the index and the catalog into the state's folder.
EXPORT SUB SAVE(state)
    DIM dir$ = state{"dir"}
    MKDIR dir$
    SEARCH.WRITEFILE(state{"ix"}, PATH.JOIN$(dir$, "index.txt"))
    DIM cat_path$ = PATH.JOIN$(dir$, "catalog.json")
    TXTWRITER cat_path$, JSON.STRINGIFY$(state{"catalog"})
ENDSUB

' About 200 characters of text around the first query word found in
' it, or the start of the text when none is.
EXPORT FUNC SNIPPET$(text$, query$)
    DIM flat$ = TRIM$(REGEX.REPLACE("\s+", text$, " "))
    DIM low$ = LCASE$(flat$)
    DIM words = SPLIT(LCASE$(REPLACE$(query$, CHR$(34), " ")), " ")
    DIM at = -1
    DIM k = 0
    FOR k = 0 TO LEN(words) - 1
        DIM w$ = TRIM$(words[k])
        IF LEN(w$) > 5 THEN w$ = LEFT$(w$, LEN(w$) - 2)
        IF w$ <> "" AND ALSO at < 0 THEN at = INSTR(low$, w$)
    NEXT k
    DIM from = 0
    IF at > 60 THEN from = at - 60
    DIM piece$ = MID$(flat$, from, 200)
    IF from > 0 THEN piece$ = "... " + piece$
    IF from + 200 < LEN(flat$) THEN piece$ = piece$ + " ..."
    RETURN piece$
ENDFUNC

' The best matches of a query, as maps with "path", "name", "score"
' and "snippet". A misspelt word finds the closest indexed one.
EXPORT FUNC FIND(state, query$, limit_n)
    DIM catalog = state{"catalog"}
    DIM opts = {"limit": limit_n, "fuzzy": 1}
    DIM hits = SEARCH.QUERY(state{"ix"}, query$, opts)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(hits) - 1
        DIM hit = hits[k]
        DIM p$ = hit{"key"}
        DIM entry = catalog{p$}
        DIM row AS MAP
        row{"path"} = p$
        row{"name"} = PATH.BASENAME$(p$)
        row{"score"} = hit{"score"}
        row{"snippet"} = SNIPPET$(entry{"text"}, query$)
        PUSH(out, row)
    
```

```

    NEXT k
    RETURN out
ENDFUNC

' The search page: a form, and the hits of query$ when there is one.
EXPORT FUNC PAGE$(state, query$)
    DIM q$ = HTMLODOM.ESCAPE$(query$)
    DIM n = SEARCH.DOCCOUNT(state{"ix"})
    DIM html$ = "<!doctype html><html><head><meta charset=utf-8>"
    html$ = html$ + "<title>Find Anything</title><style>"
    html$ = html$ + "body{font-family:sans-serif;max-width:46em;"
    html$ = html$ + "margin:2em auto;padding:0 1em}"
    html$ = html$ + "input{font-size:1.2em;width:70%}"
    html$ = html$ + ".hit{margin:1.2em 0}"
    html$ = html$ + ".path{color:#555;font-size:.85em}"
    html$ = html$ + "</style></head><body><h1>Find Anything</h1>"
    html$ = html$ + "<form><input name=q autofocus value=" + CHR$(34)
    html$ = html$ + q$ + CHR$(34) + "> <button>Search</button></form>"
    html$ = html$ + "<p>" + STR$(n) + " documents in the index.</p>"
    IF TRIM$(query$) <> "" THEN
        DIM hits = FIND(state, query$, 20)
        IF LEN(hits) = 0 THEN html$ = html$ + "<p>Nothing found.</p>"
        DIM k = 0
        FOR k = 0 TO LEN(hits) - 1
            DIM h = hits[k]
            html$ = html$ + "<div class=hit><b>"
            html$ = html$ + HTMLODOM.ESCAPE$(h{"name"}) + "</b><br>"
            html$ = html$ + HTMLODOM.ESCAPE$(h{"snippet"})
            html$ = html$ + "<div class=path>"
            html$ = html$ + HTMLODOM.ESCAPE$(h{"path"})
            html$ = html$ + "</div></div>"
        NEXT k
    ENDIF
    RETURN html$ + "</body></html>"
ENDFUNC

FUNC ShowPage(request)
    RETURN PAGE$(gState, JDWEB.PARAM$(request, "q"))
ENDFUNC

FUNC ShowHits(request)
    RETURN FIND(gState, JDWEB.PARAM$(request, "q"), 20)
ENDFUNC

' Registers the search page at / and the hits as JSON at /hits.json.
EXPORT SUB MOUNT(state)
    gState = state
    JDWEB.GET("/", ShowPage@)

```

```
JDWEB.GET("/hits.json", ShowHits@)
ENDSUB
```

How it works

1. SCAN walks through every folder and the folders below it. `walk` calls itself for each folder it meets and collects the files whose extension is on the list, with the time they last changed and their size from `FILE.STAT`. Hidden files and the `~$` files Word leaves next to an open document are skipped.
2. `TEXT$` gets the words out of a file. A Word file is read with `DOCX.READ`, which answers its paragraphs and its tables; the cells of each table row are added as one line. A PDF goes through `PDF.TEXT$`, everything else is read as text.
3. `OPEN` loads two files from the index folder: the search index that `SEARCH.WRITEFILE` saved, and `catalog.json`, which remembers for each file the time it changed, its size and the first 4000 characters of its text. Without them it starts an empty index with two fields, the words of the file name and the text, where a word in the name counts twice.
4. `PLAN` compares what `SCAN` found with the catalog. A file that is not in the catalog is new; one with another time or size has changed; one in the catalog that `SCAN` no longer found was deleted. Nothing is read yet, which is what the dry run shows.
5. `UPDATE` reads the new and changed files into the index with `SEARCH.ADD`, which replaces an older version of the same file, and takes the deleted ones out with `SEARCH.REMOVE`. A file that cannot be read, a damaged Word file for example, does not stop the run: `ReadOne$` catches the error, keeps the file in the catalog with it, and the program writes it to the log. It is tried again when it changes.
6. `FIND` asks the index with `SEARCH.QUERY`. The index ranks the files the way search engines do (the method is called BM25): a word that is rare in all files but frequent in one file makes that file a good match. With "fuzzy": 1 a word that is not in the index is replaced by the closest one within one typing mistake.
7. `SNIPPET$` finds the first query word in the stored text and cuts about 200 characters around it, so you see why the file matched.
8. `PAGE$` writes the search page as `HTML`. Everything that comes from a file or from the query goes through `HTMLDOM.ESCAPE$`, so a file name with `<` in it

cannot break the page. MOUNT puts the page at / and the same hits as JSON at /hits.json.

The paths in the index use forward slashes, C:/Users/lena/..., which Windows understands as well as the usual backslashes.

Run it

The first run reads everything; --dry-run shows the plan first:

```
jdbasic find_anything.jdb --dry-run
1840 new, 0 changed, 0 gone
would read C:/Users/lena/Documents/Budget/forecast_2027.docx
...
```

The real run ends with a line in the log, here on two lines:

```
jdbasic find_anything.jdb
2026-10-05 12:30:02 INFO [find_anything] index up to date
read=1838 dropped=0 failed=2
```

Search in the console. Each hit is its path and, below it, the piece of text around the words, here wrapped to fit the page:

```
jdbasic find_anything.jdb --search "travel costs field staff"
C:/Users/lena/Share/Controlling/Minutes/2025-11-12 team.docx
... the travel costs of the field staff are booked on cost
centre 4410 from January ...
```

Or start the page and open <http://localhost:8766/> in the browser:

```
jdbasic find_anything.jdb --serve
Search page: http://localhost:8766/
Only this computer can open it. Stop it with Ctrl+C.
```

Schedule it

The wizard runs the update every day at half past twelve, when the computer is on but you are at lunch. It also sets up a second task that starts the search page with --serve when you log on, because the recipe asks for one in its recipe.toml:

```
[[task]]
name = "search page"
schedule = "at logon"
args = ["--serve"]
```

Any recipe can declare more tasks this way; each gets a name of its own in the *AutomateWork* folder of the Task Scheduler. The page always shows the index as the last update left it.

Make it yours

The settings are the `[find_anything]` part of `work.conf`:

```
[find_anything]
folders = ["~/Documents", "S:/Controlling"]
kinds = ["txt", "md", "csv", "docx", "pdf"]
index = "~/Documents/AutomateWork/find"
language = "en"
port = 8766
```

language decides how word endings are treated: with `en`, a search for *policies* finds *policy*. Use `de` for German documents and `none` to switch it off.

Changes in the code:

- **Excel files.** Their text is in the cells. Add `"xlsx"` to `kinds`, add `XLSX` to the `IMPORT` line of the module, put this line after the first line of `TEXT$`:

```
IF ext$ = ".xlsx" THEN RETURN SheetText$(path$)
```

and this function at the end of the module:

```
FUNC SheetText$(path$)
    DIM book = XLSX.READ(path$)
    DIM names = MAP.KEYS(book)
    DIM lines = []
    DIM s = 0
    DIM r = 0
    FOR s = 0 TO LEN(names) - 1
        DIM rows = book{names[s]}
        FOR r = 0 TO LEN(rows) - 1
            PUSH(lines, JOIN(rows[r], " "))
        NEXT r
    NEXT s
```

```

NEXT s
RETURN JOIN(lines, CHR$(10))
ENDFUNC

```

- **Leave a folder out**, such as an old archive. In `Walk`, change the line `Walk(full$, kinds, out)` to

```
IF name$ <> "Archive" THEN Walk(full$, kinds, out)
```

- **More hits per page**. `PAGE$` asks for 20; change the number in `FIND(state, query$, 20)`.

When it goes wrong

- **"not a folder"** at the start: a folder in `folders` does not exist or the share is not connected. The run goes on with the others; connect the drive and run it again.
- **"not readable"** lines in the log: the file is damaged, protected with a password, or a PDF that is a scanned picture without text. The index knows the file by its name only.
- **"Port 8766 is taken"**: the page is already running, or another program uses the port. Set another port.
- **A file you just saved is not found**: the index is as old as the last update. Run the program once without options.

Balance dividend

About 40 minutes a week of searching and asking around, for anyone who works with a large shared drive.

X04 Ticket Sync

The chore

Lena's team takes requests through a ticket tracker: a cost centre to rename, a report that shows the wrong figure, a question from an auditor. Lena herself plans her day in a plain to-do file, because it is quick and it is hers. So every ticket lives twice. When a ticket is assigned to her, she copies it into the file; when she ticks a task, she opens the tracker to close the ticket; and when a colleague closes one for her, the task stays open in her file until she notices. Half an hour a week goes into keeping the two lists in step, and they are still out of step most of the time.

What you get

Ticket Sync reads your to-do file and the open tickets the tracker has for you, and brings the two together:

- a ticket assigned to you that is not in the file becomes a task;
- a task you ticked closes its ticket in the tracker;
- a task you mark with (#new) becomes a new ticket, and the file gets its number;
- a ticket someone else closed is ticked in your file;
- a ticket that was renamed in the tracker renames its task.

Everything else in the file stays as you wrote it: headings, notes and tasks without a ticket. A task belongs to a ticket when it ends with the ticket number in brackets, such as (#204).

Before you start

You need the address of your tracker, your user name there and a token, the kind of password a program uses for a tracker's REST interface. Most trackers let you create one in your profile. Put it into an environment variable once, in a command prompt:


```
setx TICKETS_TOKEN "the token from your profile"
```

New command prompts see it; the recipe asks for the token when the variable is empty. The token is never written into `work.conf`.

The recipe talks to a simple REST interface: GET `/api/tickets` with assignee and state lists the open tickets, PATCH `/api/tickets/<number>` closes one, POST `/api/tickets` creates one. Real trackers differ in their paths and field names. *Make it yours* shows where to adapt that, in three functions of the module.

The program

The program reads the settings and the file, and leaves the work to the module:

```
' Ticket Sync: keeps your to-do file and the team's ticket tracker in
' step. Tickets you are given appear as tasks, a task you tick closes
' its ticket, and a task you mark (#new) becomes a ticket. The
' settings are the [ticket_sync] part of work.conf; the token comes
' from an environment variable or is asked for, never from a file.
'
'   jdbasic ticket_sync.jdb          sync now
'   jdbasic ticket_sync.jdb --dry-run show the changes, make none

IMPORT WORKCONF, TICKETSYNC, LOGGER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "ticket_sync."
DIM home$ = "~/Documents/AutomateWork"
DIM tracker$ = WORKCONF.VALUE(cfg, key$ + "tracker", "")
DIM user$ = WORKCONF.VALUE(cfg, key$ + "user", "")
DIM own$ = home$ + "/todo.txt"
DIM todo$ = WORKCONF.FOLDER$(cfg, key$ + "todo", own$)
DIM env$ = "TICKETS_TOKEN"
DIM token_var$ = WORKCONF.VALUE(cfg, key$ + "token_env", env$)
DIM dry_run = WORKCONF.FLAG("dry-run")
DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
MKDIR PATH.JOIN$(work$, "logs")
DIM lg = LOGGER.NEW("ticket_sync")
LOGGER.TO_FILE(lg, PATH.JOIN$(work$, "logs", "ticket_sync.log"))

IF tracker$ = "" ORELSE user$ = "" THEN
    PRINT "Set tracker and user in [ticket_sync] of work.conf."
END 1
ENDIF
```

```

DIM token$ = GETENV$(token_var$)
IF token$ = "" THEN INPUT "Tracker token (not stored): "; token$

DIM text$ = ""
IF FILE.EXISTS(todo$) THEN text$ = TXTREADER$(todo$)
DIM api = TICKETSYNC.SESSION(tracker$, token$)
TRY
    DIM tickets = TICKETSYNC.OPEN_TICKETS(api, user$)
    DIM plan = TICKETSYNC.PLAN(TICKETSYNC.PARSE(text$), tickets)
    DIM said = TICKETSYNC.APPLY(plan, api, user$, dry_run)
    DIM k = 0
    FOR k = 0 TO LEN(said) - 1
        PRINT said[k]
        LOGGER.INFO(lg, said[k])
    NEXT k
    DIM fresh$ = TICKETSYNC.RENDER$(plan{"items"})
    IF fresh$ = text$ THEN
        PRINT "the to-do file is up to date"
    ELSEIF dry_run THEN
        PRINT "would write "; todo$
    ELSE
        DIM backup$ = todo$ + ".bak"
        IF FILE.EXISTS(todo$) THEN FILE.COPY(todo$, backup$, TRUE)
        TXTWRITER todo$, fresh$
        PRINT "wrote "; todo$
        LOGGER.INFO(lg, "wrote " + todo$)
    ENDIF
CATCH
    PRINT "Ticket Sync stopped: "; ERRMSG$
    LOGGER.ERROR(lg, "stopped", {"error": ERRMSG$})
END 1
ENDTRY

```

The module reads the file, asks the tracker, makes the plan and carries it out:

```

' ticketsync.jdb: the work of Ticket Sync. It reads a to-do file in
' which a task ends with its ticket number, compares it with the open
' tickets of the tracker, and plans the changes on both sides: new
' tickets become tasks, ticked tasks close their ticket, tasks marked
' (#new) become tickets, and tickets closed elsewhere are ticked.
'
'   IMPORT TICKETSYNC
'   DIM api = TICKETSYNC.SESSION("https://tracker.example", token$)
'   DIM items = TICKETSYNC.PARSE(TXTREADER$("todo.md"))
'   DIM open = TICKETSYNC.OPEN_TICKETS(api, "lena")
'   DIM plan = TICKETSYNC.PLAN(items, open)

```

```

EXPORT MODULE TICKETSYNC

IMPORT REQ

DIM gLF$ = CHR$(10)

' A REST session for the tracker: the token as bearer, three tries
' with a growing wait on a busy server.
EXPORT FUNC SESSION(base$, token$)
    DIM api = REQ.NEW(base$)
    REQ.BEARER(api, token$)
    REQ.TIMEOUT(api, 30)
    REQ.RETRIES(api, 3, 500)
    RETURN api
ENDFUNC

' The open tickets of a person as maps with id (text) and title.
EXPORT FUNC OPEN_TICKETS(api, user$)
    DIM query AS MAP
    query{"assignee"} = user$
    query{"state"} = "open"
    DIM r = REQ.GET(api, "/api/tickets", query)
    REQ.RAISE_FOR_STATUS(r)
    DIM rows = REQ.JSON(r)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM t = rows[k]
        DIM one AS MAP
        one{"id"} = IdText$(t{"id"})
        one{"title"} = t{"title"}
        PUSH(out, one)
    NEXT k
    RETURN out
ENDFUNC

FUNC IdText$(id)
    IF TYPEOF(id) = "STRING" THEN RETURN id
    RETURN STR$(INT(id))
ENDFUNC

' Every line of the to-do file as a map: kind is "task" or "text";
' a task has done, title and id ("" without a ticket, "new" for one
' still to be created).
EXPORT FUNC PARSE(text$)
    DIM rows = SPLIT(REPLACE$(text$, CHR$(13), ""), gLF$)
    IF LEN(rows) > 0 AND ALSO rows[LEN(rows) - 1] = "" THEN
        rows = DROP(-1, rows)
    
```

```

    ENDIF
    RETURN SELECT(LAMBDA row$ -> ParseLine(row$), rows)
ENDFUNC

FUNC ParseLine(row$)
    DIM item AS MAP
    item{"kind"} = "text"
    item{"line"} = row$
    DIM s$ = TRIM$(row$)
    DIM head$ = LCASE$(LEFT$(s$, 6))
    IF head$ <> "- [ ] " ANDALSO head$ <> "- [x] " THEN RETURN item
    item{"kind"} = "task"
    item{"done"} = (head$ = "- [x] ")
    DIM title$ = TRIM$(MID$(s$, 6))
    item{"id"} = ""
    DIM at = INSTR(title$, "#")
    IF at >= 0 ANDALSO ENDSWITH(title$, "#") THEN
        item{"id"} = MID$(title$, at + 2, LEN(title$) - at - 3)
        title$ = TRIM$(LEFT$(title$, at))
    ENDIF
    item{"title"} = title$
    RETURN item
ENDFUNC

' The line of a task as it goes back into the file.
EXPORT FUNC TASK_LINE$(item)
    IF item{"kind"} <> "task" THEN RETURN item{"line"}
    DIM box$ = "- [ ] "
    IF item{"done"} THEN box$ = "- [x] "
    DIM row$ = box$ + item{"title"}
    IF item{"id"} <> "" THEN row$ = row$ + " (#" + item{"id"} + ")"
    RETURN row$
ENDFUNC

' The whole to-do file from its items.
EXPORT FUNC RENDER$(items)
    RETURN JOIN(SELECT(LAMBDA it -> TASK_LINE$(it) + gLF$, items), "")
ENDFUNC

' The changes that bring the file and the tracker together, as a map
' with "remote" (close and create calls for the tracker) and "items"
' (the file after the local changes). Nothing is changed yet.
EXPORT FUNC PLAN(items, tickets)
    DIM remote = []
    DIM out = []
    DIM known AS MAP
    DIM k = 0
    FOR k = 0 TO LEN(tickets) - 1

```

```

    DIM t = tickets[k]
    DIM tid$ = t{"id"}
    known[tid$] = t{"title"}
NEXT k
DIM listed AS MAP
FOR k = 0 TO LEN(items) - 1
    DIM item = items[k]
    IF item{"kind"} = "task" ANDALSO item{"id"} = "new" THEN
        DIM title$ = item{"title"}
        PUSH(remote, Change("create", "", title$, LEN(out)))
    ELSEIF item{"kind"} = "task" ANDALSO item{"id"} <> "" THEN
        DIM id$ = item{"id"}
        listed[id$] = TRUE
        DIM is_open = MAP.EXISTS(known, id$)
        IF item{"done"} ANDALSO is_open THEN
            DIM done$ = item{"title"}
            PUSH(remote, Change("close", id$, done$, LEN(out)))
        ELSEIF NOT item{"done"} ANDALSO NOT is_open THEN
            item{"done"} = TRUE
        ELSEIF is_open THEN
            item{"title"} = known[id$]
        ENDIF
    ENDIF
    PUSH(out, item)
NEXT k
FOR k = 0 TO LEN(tickets) - 1
    DIM fresh = tickets[k]
    DIM fid$ = fresh{"id"}
    IF NOT MAP.EXISTS(listed, fid$) THEN
        DIM task AS MAP
        task{"kind"} = "task"
        task{"done"} = FALSE
        task{"title"} = fresh{"title"}
        task{"id"} = fid$
        task{"line"} = ""
        PUSH(out, task)
    ENDIF
NEXT k
DIM plan AS MAP
plan{"remote"} = remote
plan{"items"} = out
RETURN plan
ENDFUNC

FUNC Change(what$, id$, title$, index)
    DIM c AS MAP
    c{"do"} = what$
    c{"id"} = id$

```

```

    c{"title"} = title$
    c{"index"} = index
    RETURN c
ENDFUNC

' The plan's tracker calls; a created ticket's id goes into its task.
' With dry_run nothing is sent. Answers the lines it printed.
EXPORT FUNC APPLY(plan, api, user$, dry_run)
    DIM said = []
    DIM items = plan{"items"}
    DIM remote = plan{"remote"}
    DIM k = 0
    FOR k = 0 TO LEN(remote) - 1
        DIM c = remote[k]
        DIM what$ = c{"do"}
        DIM note$ = what$ + " #" + c{"id"} + " " + c{"title"}
        IF what$ = "create" THEN note$ = "create " + c{"title"}
        IF dry_run THEN
            PUSH(said, "would " + note$)
        ELSEIF what$ = "close" THEN
            DIM body AS MAP
            body{"state"} = "closed"
            DIM path$ = "/api/tickets/" + c{"id"}
            REQ.RAISE_FOR_STATUS(REQ.PATCH(api, path$, body))
            PUSH(said, note$)
        ELSE
            DIM fields AS MAP
            fields{"title"} = c{"title"}
            fields{"assignee"} = user$
            DIM r = REQ.POST(api, "/api/tickets", fields)
            REQ.RAISE_FOR_STATUS(r)
            DIM made = REQ.JSON(r)
            DIM task = items[c{"index"}]
            task{"id"} = IdText$(made{"id"})
            PUSH(said, note$ + " as #" + task{"id"})
        ENDIF
    NEXT k
    RETURN said
ENDFUNC

```

How it works

1. **A session.** SESSION creates a REQ session: the tracker's address, the token as a bearer header, a timeout of 30 seconds and three tries with a growing wait when the tracker answers that it is busy. Every call of the recipe goes through that session.

2. **The open tickets.** OPEN_TICKETS asks for the tickets of one person in the state *open* and keeps the number and the title of each. A number is kept as text, so 204 from the tracker and (#204) from the file compare as equal.
3. **The file as items.** PARSE turns every line into an item. A line that starts with - [] or - [x] is a task, with its title and, at the end, its ticket number. Any other line is text and is written back unchanged. RENDER\$ does the reverse, so a file read and written without changes comes out exactly as it went in.
4. **The plan.** PLAN walks through the tasks and decides, with the open tickets at hand. A ticked task whose ticket is still open needs a *close* call. A task marked (#new) needs a *create* call. An open task whose ticket is no longer open is ticked. Then every open ticket that no task names is added at the end of the file. The plan changes nothing yet; the calls for the tracker wait in remote.
5. **Carrying it out.** APPLY sends the calls. A *close* is a PATCH with the state *closed*; a *create* is a POST, and the number the tracker answers goes into the task. With --dry-run it only says what it would send.
6. **Writing the file.** Only when the file would change does the program write it, and it keeps the version before as `todo.txt.bak`. Each action is a line in `logs/ticket_sync.log`.
7. **Stopping cleanly.** When the tracker cannot be reached or refuses the token, REQ.RAISE_FOR_STATUS raises an error, the program prints it, logs it and ends with exit code 1. The file is not touched in that case.

Run it

Lena's file before the run:

```
- [ ] Fix VAT rounding (#201)
- [x] Update cost centres (#202)
- [ ] Ask IT for SAP access (#new)
```

A dry run shows what would happen:

```
jdbasic ticket_sync.jdb --dry-run
would create Ask IT for SAP access
would write C:\Users\lena\Documents\AutomateWork\todo.txt
```

Ticket 202 is already closed in the tracker, so ticking it needs no call. The real run creates the ticket and writes the file:

```
jdbasic ticket_sync.jdb
create Ask IT for SAP access as #206
wrote C:\Users\lena\Documents\AutomateWork\todo.txt
```

```
- [ ] Fix VAT rounding in the report (#201)
- [x] Update cost centres (#202)
- [ ] Ask IT for SAP access (#206)
- [ ] Check travel costs of Q3 (#204)
```

The first task took the title of its ticket, the new task has its number, and ticket 204, which was assigned to Lena in the meantime, is now a task. A second run right after finds nothing to do.

Schedule it

Once the token is in the environment variable, set the schedule to every 2 hours or weekdays 08:00 in the wizard. Keep the to-do file closed in your editor while the task runs, or save it first: the recipe writes the file as it read it a moment before.

Make it yours

The settings are in the part of `work.conf` that starts with `[ticket_sync]`:

```
[ticket_sync]
tracker = "https://tickets.example.com"
user = "lena"
todo = "~/Documents/AutomateWork/todo.txt"
token_env = "TICKETS_TOKEN"
```

- **Another tracker.** The paths and fields of the tracker are in three places of the module: the GET in `OPEN_TICKETS`, the PATCH and the POST in `APPLY`. For GitLab, for example, the list is `/api/v4/issues` with `assignee_username` and `state = "opened"`, and closing is a PUT on the issue with `state_event = "close"`. Change those lines and the field names `id` and `title` if your tracker calls them differently; the plan itself stays as it is.

- **A different marker.** If (#new) clashes with your notes, change the word "new" in PLAN to something else, such as "ticket".
- **Only some tickets.** Add a filter to the query in OPEN_TICKETS, for example `query{"label"} = "controlling"`, so only one kind of ticket comes into the file.
- **The same file as the dashboard.** The default is the `todo.txt` of the Personal Dashboard (M08), so the open tickets also show up there.

When it goes wrong

- **"answered 401" or "403":** the token is wrong, expired, or lacks the right to change tickets. Create a new one with write access and put it into the variable again.
- **"transport failure":** the address in `tracker` is wrong, or the tracker is only reachable inside the company network. Try the address in a browser on the same computer.
- **A task was ticked that you had not finished:** someone closed its ticket in the tracker. Untick the task first and then reopen the ticket. In the other order the next run sees a ticked task with an open ticket and closes the ticket again.
- **The file lost your last change:** you saved it while the recipe ran. The version before the run is in `todo.txt.bak`.

Balance dividend

About 30 minutes a week, and one list you can trust instead of two you have to compare.

X05 Calendar Bridge

The chore

Lena's family plans the week in a shared calendar on their phones. Her work meetings live in Outlook at the office, and the two never meet. So on Sunday evening she copies the important ones by hand, forgets the ones that move during the week, and on Tuesday her partner books the dentist into the budget review. Copying the whole work calendar is no answer either: the subjects of her meetings are nobody's business at home, and some of them are confidential.

What you get

A calendar file with your work meetings of the next two weeks as busy blocks. Each block has the right start and end and says *Busy (work)*, nothing more. Private meetings never show their subject, cancelled ones and time marked as free are left out, a tentative meeting says so. The program runs every two hours, so a meeting that moves at work moves at home as well.

The program reads the work calendar from Microsoft 365 with the access you grant it once, and writes the file. Your personal calendar subscribes to that file.

Before you start

The bridge needs to be allowed to read your work calendar. That is granted through OAuth, the sign-in that shows "this app wants to read your calendar" and asks you to agree. For that, your IT department registers the bridge once as an app in Microsoft Entra ID (the directory behind Microsoft 365). Ask for:

- an app registration with the delegated permission `Calendars.Read` and `offline_access`,
- *Allow public client flows* switched on, because the bridge signs in with a device code,
- the *Application (client) ID* of that registration.

The client id goes into `work.conf`. It is no secret; it only names the app. Many companies let you register such an app yourself, and it costs nothing. If your IT registers it as a confidential app with a client secret, that secret goes into the environment variable `AUTOMATEWORK_CALENDAR_SECRET`, never into a file.

On the personal side you need a calendar that subscribes to a file. Thunderbird does this with a file on your computer; a phone calendar needs the file in a cloud folder it can reach. If your calendar can only import, import the file once a week instead.

Watch out

After you sign in, the bridge keeps a *refresh token*: a key that lets it read your calendar without asking you again. It is kept in `.automatework/calendar_token.json` in your user folder, outside the work folder, so it is not copied with your settings or picked up by a backup of AutomateWork. Treat this file like a password. Delete it to sign the bridge out, or remove the app in your account settings. X14 shows how to keep such a token in the protected store of Windows instead of a file.

The program

The program reads the settings, signs in or uses the kept tokens, reads the calendar and writes the file:

```
' Calendar Bridge: the meetings of your work calendar as busy blocks
' in your personal calendar, which then shows when you are booked but
' not what about. It reads Microsoft 365 through OAuth and writes a
' calendar file that a personal calendar subscribes to. The settings
' are the [calendar_bridge] part of work.conf.
'
'   jdbasic calendar_bridge.jdb --login      sign in once
'   jdbasic calendar_bridge.jdb             write the busy blocks
'   jdbasic calendar_bridge.jdb --dry-run   show what would change

IMPORT WORKCONF, CALBRIDGE, DT, LOGGER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "calendar_bridge."
DIM client_id$ = WORKCONF.VALUE(cfg, key$ + "client_id", "")
IF client_id$ = "" THEN THROW "Set client_id in [calendar_bridge]."
DIM tenant$ = WORKCONF.VALUE(cfg, key$ + "tenant", "organizations")
DIM authority$ = WORKCONF.VALUE(cfg, key$ + "authority", _
    CALBRIDGE.AUTHORITY$(tenant$))
DIM api$ = WORKCONF.VALUE(cfg, key$ + "api", _
    "https://graph.microsoft.com/v1.0")
DIM days = WORKCONF.VALUE(cfg, key$ + "days", 14)
DIM label$ = WORKCONF.VALUE(cfg, key$ + "label", "Busy (work)")
```

```

DIM show_titles = WORKCONF.VALUE(cfg, key$ + "show_titles", FALSE)
DIM out_home$ = "~/Documents/AutomateWork/calendar/work-busy.ics"
DIM output$ = WORKCONF.FOLDER$(cfg, key$ + "output", out_home$)
DIM token_home$ = "~/.automatework/calendar_token.json"
DIM token_file$ = WORKCONF.FOLDER$(cfg, key$ + "token_file", _
    token_home$)
DIM secret$ = GETENV$("AUTOMATEWORK_CALENDAR_SECRET")
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM lg = LOGGER.NEW("calendar_bridge")
LOGGER.TO_CONSOLE(lg, FALSE)
IF NOT dry_run THEN
    MKDIR PATH.DIRNAME$(output$)
    DIM log$ = PATH.JOIN$(PATH.DIRNAME$(output$), "bridge.log")
    LOGGER.TO_FILE(lg, log$, 1000000, 2)
ENDIF

DIM app = CALBRIDGE.CLIENT(client_id$, secret$, authority$)
CALBRIDGE.KEEP(app, token_file$)
IF WORKCONF.FLAG("login") THEN
    IF CALBRIDGE.LOGIN(app, 900) THEN
        PRINT "Signed in. The tokens are kept in"
        PRINT " "; token_file$
    ENDIF
END
ENDIF
IF NOT CALBRIDGE.SIGNED_IN(app) THEN
    THROW "Not signed in yet: run calendar_bridge.jdb --login once."
ENDIF

DIM t_from = DT.STARTOF(DT.NOW(), "day")
DIM t_to = t_from + days * 86400
DIM events = []
TRY
    events = CALBRIDGE.FETCH(app, api$, t_from, t_to, 2000)
CATCH
    DIM why$ = ERRMSG$
    CALBRIDGE.SAVE(app)
    LOGGER.ERROR(lg, "calendar not read: " + why$)
    THROW why$
ENDTRY
CALBRIDGE.SAVE(app)

DIM blocks = CALBRIDGE.BLOCKS(events, label$, show_titles)
DIM change = CALBRIDGE.CHANGES(output$, blocks)
IF dry_run THEN
    DIM local = DT.LOCAL_OFFSET()
    DIM k = 0

```

```

FOR k = 0 TO LEN(blocks) - 1
    DIM b = blocks[k]
    DIM when$ = DT.FMT$(b{"start"}, "%a %d.%m. %H:%M", local)
    IF b{"all_day"} THEN
        when$ = DT.FMT$(b{"start"}, "%a %d.%m.", 0) + " all day"
    ENDIF
    PRINT when$; " "; b{"summary"}
NEXT k
PRINT change{"added"}; " new, "; change{"changed"}; _
    " changed, "; change{"removed"}; " gone; nothing written"
END
ENDIF

TXTWRITER output$, CALBRIDGE.ICS$(blocks)
LOGGER.INFO(lg, "busy blocks written", {"blocks": LEN(blocks), _
    "new": change{"added"}, "changed": change{"changed"}, _
    "gone": change{"removed"}})

```

The module does the talking to Microsoft and the turning into blocks:

```

' calbridge.jdb: the work of the Calendar Bridge. It signs in to a
' Microsoft 365 work calendar through OAuth, reads the events of the
' coming days, turns them into busy blocks that do not say what the
' meetings are about, and writes them as a calendar file that a
' personal calendar can subscribe to.
'
'   IMPORT CALBRIDGE
'   DIM authority$ = CALBRIDGE.AUTHORITY$("organizations")
'   DIM app = CALBRIDGE.CLIENT(client_id$, "", authority$)
'   CALBRIDGE.KEEP(app, "C:/Users/lena/.automatework/calendar.json")
'   DIM api$ = "https://graph.microsoft.com/v1.0"
'   DIM events = CALBRIDGE.FETCH(app, api$, t_from, t_to, 2000)
'   DIM blocks = CALBRIDGE.BLOCKS(events, "Busy (work)", FALSE)
'   TXTWRITER "work-busy.ics", CALBRIDGE.ICS$(blocks)

EXPORT MODULE CALBRIDGE

IMPORT OAUTH, CACHE, RETRY, REQ, DT, ICAL

DIM gScope$ = "offline_access Calendars.Read"
DIM gApp AS MAP
DIM gSess AS MAP

' — signing in —————

' The sign-in address of a Microsoft 365 tenant: "organizations" for
' any work account, or the tenant's own id.

```

```

EXPORT FUNC AUTHORITY$(tenant$)
    DIM root$ = "https://login.microsoftonline.com/"
    RETURN root$ + tenant$ + "/oauth2/v2.0"
ENDFUNC

' An OAuth client for the authority's endpoints. A client registered
' as a public app needs no secret; pass "" then.
EXPORT FUNC CLIENT(client_id$, secret$, authority$)
    DIM auth$ = authority$ + "/authorize"
    DIM token$ = authority$ + "/token"
    DIM device$ = authority$ + "/devicecode"
    DIM app = OAUTH.NEW(client_id$, secret$, auth$, token$, device$)
    OAUTH.SETOPT(app, "scope", gScope$)
    RETURN app
ENDFUNC

' Keeps the client's tokens in token_file$: tokens already there are
' taken now, and SAVE writes the newest ones back.
EXPORT SUB KEEP(app, token_file$)
    DIM store = 0
    IF FILE.EXISTS(token_file$) THEN
        store = CACHE.LOAD(token_file$)
    ELSE
        store = CACHE.NEW()
    ENDIF
    OAUTH.STORE(app, store, "calendar")
    app{"token_file"} = token_file$
ENDSUB

' Writes the client's tokens to its token file.
EXPORT SUB SAVE(app)
    DIM path$ = app{"token_file"}
    MKDIR PATH.DIRNAME$(path$)
    CACHE.SAVE(OAUTH.CACHEOF(app), path$)
ENDSUB

' Whether the client has a refresh token, so it can run unattended.
EXPORT FUNC SIGNED_IN(app)
    DIM refresh$ = app{"refresh_token"}
    RETURN refresh$ <> ""
ENDFUNC

' Signs in with a code shown here and typed into a browser on any
' device, waits at most max_seconds, and saves the tokens.
EXPORT FUNC LOGIN(app, max_seconds)
    DIM shown = OAUTH.DEVICESTART(app, gScope$)
    DIM msg$ = shown{"message"}
    IF msg$ = "" THEN

```

```

        msg$ = "Open " + shown{"verification_uri"}
        msg$ = msg$ + " and enter the code " + shown{"user_code"}
    ENDIF
    PRINT msg$
    DIM token$ = OAUTH.DEVICE POLL(app, max_seconds)
    SAVE(app)
    RETURN token$ <> ""
ENDFUNC

' — reading the calendar —————

FUNC Transient(message$)
    RETURN STARTSWITH(message$, "transient")
ENDFUNC

' One page of the calendar API. A busy or unreachable server raises a
' "transient" error, which RETRY tries again; anything else is final.
FUNC GetPage(url$)
    DIM r = OAUTH.GET(gApp, gSess, url$)
    DIM status = r{"status"}
    DIM body$ = r{"body"}
    IF status = 0 ORELSE status = 429 ORELSE status >= 500 THEN
        THROW "transient: the calendar answered " + STR$(status)
    ENDIF
    IF status <> 200 THEN
        DIM said$ = STR$(status) + ": " + LEFT$(body$, 200)
        THROW "The calendar answered " + said$
    ENDIF
    RETURN JSON.PARSE$(body$)
ENDFUNC

FUNC Text$(row, name$)
    IF NOT MAP.EXISTS(row, name$) THEN RETURN ""
    DIM v = row{name$}
    IF TYPEOF(v) <> "STRING" THEN RETURN ""
    RETURN v
ENDFUNC

FUNC Flag(row, name$)
    IF NOT MAP.EXISTS(row, name$) THEN RETURN FALSE
    DIM v = row{name$}
    IF TYPEOF(v) <> "BOOLEAN" THEN RETURN FALSE
    RETURN v
ENDFUNC

' An event of the API as a map: id, subject, start and end as DT
' instants, all_day, cancelled, show_as and private.
FUNC Event(row)

```

```

DIM ev AS MAP
ev{"id"} = Text$(row, "id")
ev{"subject"} = Text$(row, "subject")
DIM st = row{"start"}
DIM en = row{"end"}
DIM st$ = st{"dateTime"}
DIM en$ = en{"dateTime"}
ev{"start"} = DT.PARSE(st$ + "Z")
ev{"end"} = DT.PARSE(en$ + "Z")
ev{"all_day"} = Flag(row, "isAllDay")
ev{"cancelled"} = Flag(row, "isCancelled")
ev{"show_as"} = LCASE$(Text$(row, "showAs"))
DIM sens$ = LCASE$(Text$(row, "sensitivity"))
ev{"private"} = sens$ = "private" ORELSE sens$ = "confidential"
RETURN ev
ENDFUNC

' The events between two instants, following the API's pages. Each
' page is tried up to four times, waiting delay_ms and then twice as
' long each time.
EXPORT FUNC FETCH(app, api$, t_from, t_to, delay_ms)
    gApp = app
    gSess = OAUTH.SESSION(app, api$)
    DIM utc$ = "outlook.timezone=" + CHR$(34) + "UTC" + CHR$(34)
    REQ.HEADER(gSess, "Prefer", utc$)
    DIM q AS MAP
    q{"startDateTime"} = DT.ISO$(t_from)
    q{"endDateTime"} = DT.ISO$(t_to)
    q{"$top"} = 100
    DIM url$ = api$ + "/me/calendarView?" + REQ.QUERY$(q)
    DIM opts = {"attempts": 4, "delay": delay_ms}
    opts{"retry_if"} = Transient@
    DIM events = []
    DIM pages = 0
    DO WHILE url$ <> "" AND ALSO pages < 50
        DIM report = RETRY.ATTEMPT(GetPage@, url$, opts)
        IF NOT report{"ok"} THEN
            DIM why$ = report{"error"}
            THROW why$
        ENDIF
        DIM page = report{"value"}
        DIM rows = page{"value"}
        DIM k = 0
        FOR k = 0 TO LEN(rows) - 1
            PUSH(events, Event(rows[k]))
        NEXT k
        url$ = Text$(page, "@odata.nextLink")
        pages = pages + 1
    
```



```

    LOOP
    RETURN events
ENDFUNC

' — busy blocks —————

' The events that make you busy, as blocks with uid, summary, start,
' end and all_day. Cancelled and free events are left out. The summary
' is label$, or the subject when show_titles is on and the meeting is
' not private; a tentative meeting says so.
EXPORT FUNC BLOCKS(events, label$, show_titles)
    DIM busy = FILTER(LAMBDA ev -> Busy(ev), events)
    RETURN SELECT(LAMBDA USE(label$, show_titles) ev -> _
        Block(ev, label$, show_titles), busy)
ENDFUNC

' Whether an event makes you busy.
FUNC Busy(ev)
    DIM show$ = ev{"show_as"}
    RETURN NOT ev{"cancelled"} ANDALSO show$ <> "free"
ENDFUNC

' One event as a block.
FUNC Block(ev, label$, show_titles)
    DIM b AS MAP
    DIM id$ = ev{"id"}
    DIM hash$ = LEFT$(CODEC.SHA256$(id$), 24)
    b{"uid"} = hash$ + "@calendar-bridge"
    DIM summary$ = label$
    IF show_titles ANDALSO NOT ev{"private"} THEN
        summary$ = ev{"subject"}
    ENDIF
    IF ev{"show_as"} = "tentative" THEN
        summary$ = summary$ + " (tentative)"
    ENDIF
    b{"summary"} = summary$
    b{"start"} = ev{"start"}
    b{"end"} = ev{"end"}
    b{"all_day"} = ev{"all_day"}
    RETURN b
ENDFUNC

' The blocks as the text of an .ics calendar.
EXPORT FUNC ICS$(blocks)
    DIM cal = ICAL.NEW("-//AutomateWork//Calendar Bridge//EN")
    DIM k = 0
    FOR k = 0 TO LEN(blocks) - 1
        DIM b = blocks[k]

```

```

    DIM ev = 0
    IF b{"all_day"} THEN
        DIM days = ROUND((b{"end"} - b{"start"}) / 86400)
        IF days < 1 THEN days = 1
        ev = ICAL.ADDALLDAY(cal, b{"summary"}, b{"start"}, days)
        ICAL.SETPROP(cal, ev, "TRANSP", "OPAQUE")
    ELSE
        DIM s$ = b{"summary"}
        ev = ICAL.ADDEVENT(cal, s$, b{"start"}, b{"end"})
    ENDIF
    ICAL.SETPROP(cal, ev, "UID", b{"uid"})
NEXT k
RETURN ICAL.TEXT$(cal)
ENDFUNC

FUNC Signature$(start, finish, summary$)
    RETURN STR$(start) + "|" + STR$(finish) + "|" + summary$
ENDFUNC

' How the blocks differ from the calendar file at path$: the numbers
' "added", "changed" and "removed".
EXPORT FUNC CHANGES(path$, blocks)
    DIM before AS MAP
    IF FILE.EXISTS(path$) THEN
        DIM cal = ICAL.PARSEFILE(path$)
        DIM evs = ICAL.EVENTS(cal)
        DIM k = 0
        FOR k = 0 TO LEN(evs) - 1
            DIM e = evs[k]
            DIM uid$ = ICAL.PROP$(cal, e, "UID")
            DIM sum$ = ICAL.PROP$(cal, e, "SUMMARY")
            DIM t_a = ICAL.STARTAT(cal, e)
            DIM t_b = ICAL.ENDAT(cal, e)
            before{uid$} = Signature$(t_a, t_b, sum$)
        NEXT k
    ENDIF
    DIM added = 0
    DIM changed = 0
    DIM seen AS MAP
    DIM j = 0
    FOR j = 0 TO LEN(blocks) - 1
        DIM b = blocks[j]
        DIM id$ = b{"uid"}
        seen{id$} = TRUE
        DIM now$ = Signature$(b{"start"}, b{"end"}, b{"summary"})
        IF NOT MAP.EXISTS(before, id$) THEN
            added = added + 1
        ELSEIF before{id$} <> now$ THEN

```

```

        changed = changed + 1
    ENDIF
NEXT j
DIM gone = FILTER(LAMBDA USE(seen) id$ -> _
    NOT MAP.EXISTS(seen, id$), MAP.KEYS(before))
DIM removed = LEN(gone)
RETURN {"added": added, "changed": changed, "removed": removed}
ENDFUNC

```

How it works

1. CLIENT makes an OAuth client from `lib/oauth.jdb` with the three addresses of the sign-in service: one to ask for a device code, one to trade the code for tokens, one to authorize in the browser.
2. KEEP connects the client to the token file. OAUTH.STORE takes any tokens already in the file, and every new token the client gets is put into the same store; SAVE writes it back with CACHE.SAVE. The program saves after every run, also after one that failed, because the sign-in service hands out a new refresh token from time to time and the old one stops working.
3. LOGIN runs the device code flow: OAUTH.DEVICESTART asks for a short code and an address, the program prints both, you open the address on any device and type the code, and OAUTH.DEVICE POLL waits until you have agreed. No password passes through the program.
4. FETCH asks the Microsoft Graph API for the events between two instants at `/me/calendarView`. The Prefer header asks for all times in UTC, so the bridge never has to know about time zones. OAUTH.GET puts the access token on each request and gets a new one with the refresh token when it has run out.
5. The API answers in pages of up to 100 events, each with the address of the next page in `@odata.nextLink`. FETCH follows them. Each page goes through RETRY.ATTEMPT: when the server answers 429 (too many requests) or a 5xx error, or does not answer at all, GetPage raises an error that starts with *transient*, and Transient tells RETRY to try again after 2, 4 and 8 seconds. Any other answer, such as a revoked sign-in, stops at once with the message of the server.
6. Event turns each API event into a small map with the times as DT instants (seconds since 1970 in UTC). BLOCKS keeps the events that make you busy with FILTER and the function Busy, and SELECT turns each one into a block

with Block, which decides what your family sees. The id of each block is a hash of the event's id, so the same meeting has the same id in every run and a calendar moves it instead of adding a second one.

7. ICS\$ writes the blocks with ICAL: a meeting as an event from start to end, an all-day event as a day, marked busy.
8. CHANGES reads the file of the last run and counts the blocks that are new, that moved or that went, for the log and the dry run.

Run it

Sign in once:

```
jdbasic calendar_bridge.jdb --login
To sign in, use a web browser to open the page
https://microsoft.com/devicelogin and enter the code KJ7Q2ZPRT to
authenticate.
Signed in. The tokens are kept in
C:\Users\lena\.automatework/calendar_token.json
```

Then look before you write, and write:

```
jdbasic calendar_bridge.jdb --dry-run
Mon 05.10. 09:00 Busy (work)
Mon 05.10. 14:00 Busy (work)
Tue 06.10. 13:00 Busy (work) (tentative)
Thu 08.10. all day Busy (work)
4 new, 0 changed, 0 gone; nothing written
jdbasic calendar_bridge.jdb
```

The log `bridge.log` next to the calendar file has a line per run.

Schedule it

The wizard runs the bridge every two hours while you are logged on. A run takes a second or two. If the computer was off, the next run catches up, because each run writes the whole two weeks again.

Make it yours

The settings are the `[calendar_bridge]` part of `work.conf`:

```
[calendar_bridge]
client_id = "8f2c51d7-0a4e-4c55-9b1e-3d6e2a7f9c10"
tenant = "organizations"
days = 14
label = "Busy (work)"
show_titles = false
output = "~/OneDrive/Calendar/work-busy.ics"
```

With `show_titles = true` your blocks carry the meeting subjects, except for private and confidential meetings, which always stay *Busy (work)*.

Changes in the code:

- **Only the core hours.** To leave out early and late meetings, add a condition in *Busy* next to the one for free time, for example with `DT.FMT$(ev{"start"}, "%H", 2)` for the hour in UTC+2.
- **Mark home office days.** If you mark them with an all-day event called *Home office*, let Block give such events their subject even when `show_titles` is off. Put this line before the one that checks for "tentative":

```
IF ev{"subject"} = "Home office" THEN summary$ = "Home office"
```

- **Another calendar service.** CLIENT takes the address of any OAuth server, and the settings `authority` and `api` override the Microsoft ones. A different service also answers in a different shape, so *Event* and the page loop in *FETCH* would change with it.

When it goes wrong

- **"Set client_id in [calendar_bridge]":** the client id from your IT is missing in `work.conf`.
- **"Not signed in yet":** no token file, or it was deleted. Run `--login` once.
- **"OAUTH: invalid_grant":** the sign-in has run out or was revoked, after a password change for example. Run `--login` again.

- **"The calendar answered 403":** the app registration lacks the `Calendars.Read` permission, or your company requires an admin to approve it. Ask your IT.
- **The personal calendar shows old blocks:** many calendar apps read a subscribed calendar only every few hours. The file itself is current; look at the time in `bridge.log`.

Balance dividend

About 25 minutes a week of copying meetings, and no more evenings booked twice.

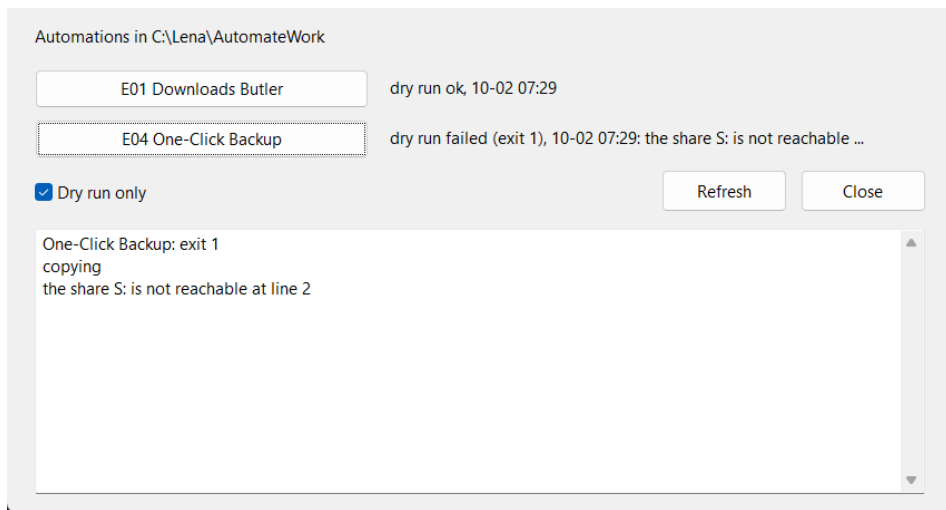
X06 Work Cockpit

The chore

Lena's automations run in the background, and that is the point. Now and then she wants one of them right away: the backup before she leaves for a week, the downloads tidied before a screen share, the status mail a day early. Each time she opens a command prompt, finds the folder, remembers the name of the program and whether it takes `--dry-run`. And when a colleague asks whether last night's backup went through, she opens a log file and searches. It is a few minutes each time, several times a week, and it is the part of her automations her colleagues never see.

What you get

A window with one button per automation in your work folder. Next to each button it shows when the automation ran last from this window and how it ended. A click starts it, as a dry run when the box is ticked, and the output appears at the bottom.



The cockpit after a dry run of the Downloads Butler and a failed backup

Every start lands in `logs/cockpit.log` of the work folder, with the time, the automation, the kind of run, the exit code and the last line of the output.

Before you start

The recipes must be installed by the wizard, so that each has its folder with a `recipe.toml` under `recipes` in your work folder. The cockpit needs the forms release of `jdBasic`, the one the wizard needs. Nothing else: it reads the work folder and the `jdbasic` program from the `[paths]` part of `work.conf`.

The program

The program builds the window, with one button and one label per automation, and handles the clicks:

```
' Work Cockpit: a window with one button per installed automation
' and, next to it, when it ran last and how it ended. A click starts
' the automation, as a dry run when the box is ticked. Every start
' lands in logs/cockpit.log of the work folder. Needs the FORMS build.
'
'   jdbasic work_cockpit.jdb           open the cockpit
'   jdbasic work_cockpit.jdb --selftest try it on a made-up work
'                                       folder and close
'   ... --shots <folder>               keep a picture of the test
'   ... --work <folder>                the made-up folder there
'   ... --jdbasic <path>               the jdbasic program to use

IMPORT WORKCONF, COCKPIT

DIM gSelfTest = "--selftest" IN OS.ARGS()
DIM gShots$ = WORKCONF.ARG$("shots")
DIM gTitle$ = "Work Cockpit"
DIM LF$ = CHR$(10)
DIM gJdbasic$ = WORKCONF.ARG$("jdbasic")
DIM gWork$ = ""
DIM gConf$ = ""
DIM gTries = 1
IF gSelfTest THEN
    IF gJdbasic$ = "" THEN THROW "The self test needs --jdbasic."
    gWork$ = COCKPIT.SAMPLE_WORK$(gJdbasic$, WORKCONF.ARG$("work"))
    gConf$ = PATH.JOIN$(gWork$, "config", "work.conf")
ELSE
    gConf$ = WORKCONF.CONFIG_PATH$()
    DIM cfg = WORKCONF.SETTINGS(gConf$)
    DIM home$ = "~/Documents/AutomateWork"
    gWork$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
    IF gJdbasic$ = "" THEN
        gJdbasic$ = WORKCONF.VALUE(cfg, "paths.jdbasic", "jdbasic")
    ENDIF
```



```

    gTries = WORKCONF.VALUE(cfg, "work_cockpit.tries", 1)
ENDIF
DIM gLog$ = PATH.JOIN$(gWork$, "logs", "cockpit.log")
DIM gList = COCKPIT.INSTALLED(gWork$, "X06")
DIM gShown = MIN([LEN(gList), 12])

' — the window —————

DIM top = 40 + gShown * 36
DIM frm = FORM.CREATE(gTitle$, 680, top + 250, "CKP")
DIM head$ = "Automations in " + gWork$
DIM lblHead = FORM.LABEL(frm, "lblHead", head$, 20, 12, 640, 22)
DIM run_buttons = []
DIM state_labels = []
DIM k = 0
FOR k = 0 TO gShown - 1
    DIM entry = gList[k]
    DIM y = 44 + k * 36
    DIM caption$ = entry{"id"} + " " + entry{"name"}
    DIM b_name$ = "btnRun" + STR$(k)
    DIM l_name$ = "lblState" + STR$(k)
    DIM b = FORM.BUTTON(frm, b_name$, caption$, 20, y, 240, 28)
    DIM l = FORM.LABEL(frm, l_name$, "", 275, y + 5, 385, 22)
    PUSH(run_buttons, b)
    PUSH(state_labels, l)
NEXT k
DIM chkDry = FORM.CHECKBOX(frm, "chkDry", "Dry run only", _
    20, top + 8, 200, 24)
DIM btnRefresh = FORM.BUTTON(frm, "btnRefresh", "Refresh", _
    470, top + 4, 90, 30)
DIM btnClose = FORM.BUTTON(frm, "btnClose", "Close", _
    570, top + 4, 90, 30)
DIM txtOut = FORM.TEXTBOX(frm, "txtOut", "", _
    20, top + 46, 640, 190, TRUE)
DIM tmrSelf = FORM.TIMER(frm, "tmrSelf", 0)
DIM gTicks = 0
DIM gFails = 0

SUB ShowStates()
    DIM rows = COCKPIT.READ_LOG(gLog$)
    DIM i = 0
    FOR i = 0 TO gShown - 1
        DIM e = gList[i]
        DIM last = COCKPIT.LAST(rows, e{"id"})
        FORM.SET(state_labels[i], "TEXT", COCKPIT.STATUS$(last))
    NEXT i
ENDSUB

```

```

SUB RunSlot(slot)
    DIM e = gList[slot]
    DIM dry = FORM.GET(chkDry, "CHECKED")
    FORM.SET(txtOut, "TEXT", "Running " + e{"name"})
    DIM r = COCKPIT.RUN(gJdbasic$, e, gConf$, dry, gTries)
    COCKPIT.APPEND_LOG(gLog$, COCKPIT.LOG_LINE$(NOW(), e, dry, r))
    DIM top$ = e{"name"} + ": exit " + STR$(r{"exit"})
    IF r{"tries"} > 1 THEN
        top$ = top$ + " after " + STR$(r{"tries"}) + " starts"
    ENDIF
    DIM body$ = top$ + LF$ + COCKPIT.CLEAN$(r{"output"})
    FORM.SET(txtOut, "TEXT", REPLACE$(body$, LF$, CHR$(13) + LF$))
    ShowStates()
ENDSUB

' — events —————

SUB RunClick(e)
    DIM ev = e[0]
    DIM name$ = ev{"name"}
    RunSlot(CINT(VAL(MID$(name$, 6, LEN(name$)))))
ENDSUB
FOR k = 0 TO gShown - 1
    DIM handler$ = "BTNRUN" + STR$(k) + "_CLICK"
    ON handler$ CALL RunClick
NEXT k

SUB CKP_LOAD(e)
    ShowStates()
    IF gShown = 0 THEN FORM.SET(txtOut, "TEXT", "Nothing installed.")
    IF gSelfTest THEN FORM.SET(tmrSelf, "INTERVAL", 200)
ENDSUB

SUB BTNREFRESH_CLICK(e)
    ShowStates()
ENDSUB

SUB BTNCLOSE_CLICK(e)
    FORM.CLOSE(frm)
ENDSUB

' — the self test —————

SUB Expect(ok, what$)
    IF NOT ok THEN
        gFails = gFails + 1
        PRINT "FAIL: "; what$
    ENDIF

```

```

ENDSUB

SUB TMRSELF_TICK(e)
    gTicks = gTicks + 1
    IF gTicks = 1 THEN
        Expect(gShown = 2, "two automations, the cockpit left out")
        DIM first$ = FORM.GET(state_labels[0], "TEXT")
        Expect(first$ = "not run yet", "nothing ran yet")
        FORM.SET(chkDry, "CHECKED", TRUE)
        FORM.SET(run_buttons[0], "VALUE", TRUE)
    ELSEIF gTicks = 2 THEN
        DIM said$ = FORM.GET(txtOut, "TEXT")
        Expect(INSTR(said$, "would sort") >= 0, "a dry run ran")
        DIM ok$ = FORM.GET(state_labels[0], "TEXT")
        Expect(STARTSWITH(ok$, "dry run ok"), "and is shown as ok")
        FORM.SET(run_buttons[1], "VALUE", TRUE)
    ELSEIF gTicks = 3 THEN
        DIM bad$ = FORM.GET(state_labels[1], "TEXT")
        Expect(INSTR(bad$, "failed") >= 0, "a failure is shown")
        DIM n = LEN(COCKPIT.READ_LOG(gLog$))
        Expect(n = 2, "both starts are logged")
        IF gShots$ <> "" THEN
            MKDIR gShots$
            DIM shot$ = PATH.JOIN$(gShots$, "x06_cockpit.png")
            DIM got = OS.SCREENSHOT(shot$, "client", gTitle$)
        ENDIF
        FORM.CLOSE(frm)
    ENDIF
ENDSUB

FORM.RUN(frm)
IF gSelfTest THEN
    OS.EXEC("cmd", ["/c", "rmdir", "/s", "/q", gWork$])
    IF gFails > 0 THEN THROW STR$(gFails) + " checks failed"
    PRINT "ALL COCKPIT WINDOW TESTS PASSED"
ENDIF

```

The module finds the automations, starts them, and reads and writes the log:

```

' cockpit.jdb: the work of the Work Cockpit. It finds the automations
' installed in the work folder, starts one of them, keeps a log of
' every start, and reads from that log when each one ran last and how
' it ended.
'
'   IMPORT COCKPIT
'   DIM list = COCKPIT.INSTALLED("C:/Users/lena/AutomateWork", "X06")
'   DIM r = COCKPIT.RUN("jdbbasic", list[0], conf$, TRUE, 1)

```

```

'   DIM line$ = COCKPIT.LOG_LINE$(NOW(), list[0], TRUE, r)
'   COCKPIT.APPEND_LOG(log_file$, line$)

EXPORT MODULE COCKPIT

IMPORT CONF

DIM gTab$ = CHR$(9)
DIM gLF$ = CHR$(10)

FUNC Text$(m, key$)
    IF NOT MAP.EXISTS(m, key$) THEN RETURN ""
    DIM v$ = "" + m{key$}
    RETURN v$
ENDFUNC

' The recipes in work$/recipes, sorted by id, as maps with "id",
' "name", "summary" and "program" (its full path). The recipe with the
' id own$ is left out.
EXPORT FUNC INSTALLED(work$, own$)
    DIM out = []
    DIM root$ = PATH.JOIN$(work$, "recipes")
    IF NOT FILE.ISDIR(root$) THEN RETURN out
    DIM names = SORT(DIR$(PATH.JOIN$(root$, "*")))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM folder_name$ = names[k]
        DIM dir$ = PATH.JOIN$(root$, folder_name$)
        DIM toml$ = PATH.JOIN$(dir$, "recipe.toml")
        IF FILE.EXISTS(toml$) THEN
            DIM r = CONF.TOML(toml$)
            DIM id$ = Text$(r, "id")
            IF id$ <> own$ AND ALSO Text$(r, "program") <> "" THEN
                DIM e AS MAP
                e{"id"} = id$
                e{"name"} = Text$(r, "name")
                e{"summary"} = Text$(r, "summary")
                e{"program"} = PATH.JOIN$(dir$, Text$(r, "program"))
                PUSH(out, e)
            ENDIF
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' The arguments that start a recipe with the settings file conf$.
EXPORT FUNC RUN_ARGS(entry, conf$, dry_run)
    DIM args = [entry{"program"}, "--config", conf$]

```

```

    IF dry_run THEN PUSH(args, "--dry-run")
    RETURN args
ENDFUNC

' A program's output without colour codes and carriage returns.
EXPORT FUNC CLEAN$(output$)
    DIM t$ = REGEX.REPLACE(CHR$(27) + "[0-9;]*m", output$, "")
    t$ = REGEX.REPLACE("Error #[0-9]+: ", t$, "")
    RETURN TRIM$(REPLACE$(t$, CHR$(13), ""))
ENDFUNC

' The last line of a program's output that is not empty.
EXPORT FUNC LAST_LINE$(output$)
    DIM lines = SPLIT(CLEAN$(output$), gLF$)
    IF LEN(lines) = 0 THEN RETURN ""
    DIM last$ = TRIM$(lines[LEN(lines) - 1])
    RETURN last$
ENDFUNC

' Starts a recipe and waits for it. A run that fails is started again
' up to tries times, a second apart. Answers "ok", "exit", "output"
' and "tries", the number of starts.
EXPORT FUNC RUN(jdbasic$, entry, conf$, dry_run, tries)
    DIM args = RUN_ARGS(entry, conf$, dry_run)
    DIM out AS MAP
    DIM n = 0
    DO
        n = n + 1
        DIM r = OS.EXEC(jdbasic$, args)
        out{"exit"} = r{"EXIT_CODE"}
        out{"output"} = r{"OUTPUT"}
        IF r{"EXIT_CODE"} = 0 ORELSE n >= tries THEN EXITDO
        SLEEP 1000
    LOOP
    out{"ok"} = out{"exit"} = 0
    out{"tries"} = n
    RETURN out
ENDFUNC

' One line of the log: time, id, run or dry, exit code and the last
' line of the output, separated by tabs.
EXPORT FUNC LOG_LINE$(when, entry, dry_run, result)
    DIM mode$ = "run"
    IF dry_run THEN mode$ = "dry"
    DIM msg$ = REPLACE$(LAST_LINE$(result{"output"}), gTab$, " ")
    DIM t$ = FORMAT_DATE(when, "%Y-%m-%d %H:%M")
    DIM code$ = STR$(result{"exit"})
    DIM parts = [t$, entry{"id"}, mode$, code$, msg$]

```

```

    RETURN JOIN(parts, gTab$)
ENDFUNC

' Adds a line to the log. A log that another program holds open is
' tried again three times before it counts as an error.
EXPORT SUB APPEND_LOG(log_file$, line$)
    MKDIR PATH.DIRNAME$(log_file$)
    DIM tries = 0
    DO
        tries = tries + 1
        TRY
            TXTWRITER log_file$, line$ + gLF$, TRUE
            EXITDO
        CATCH
            DIM why$ = "Cannot write " + log_file$ + ": " + ERRMSG$
            IF tries >= 3 THEN THROW why$
            SLEEP 200
        ENDTRY
    LOOP
ENDSUB

' The log as maps with "time", "id", "mode", "exit" and "message";
' lines that do not have five parts are skipped.
EXPORT FUNC READ_LOG(log_file$)
    DIM out = []
    IF NOT FILE.EXISTS(log_file$) THEN RETURN out
    DIM text$ = REPLACE$(TXTREADER$(log_file$), CHR$(13), "")
    DIM lines = SPLIT(text$, gLF$)
    DIM whole = FILTER(LAMBDA s$ -> LEN(SPLIT(s$, gTab$)) = 5, lines)
    RETURN SELECT(LAMBDA s$ -> LogRow(s$), whole)
ENDFUNC

' One line of the log as a map.
FUNC LogRow(line$)
    DIM parts = SPLIT(line$, gTab$)
    DIM row AS MAP
    row{"time"} = parts[0]
    row{"id"} = parts[1]
    row{"mode"} = parts[2]
    row{"exit"} = CINT(VAL(parts[3]))
    row{"message"} = parts[4]
    RETURN row
ENDFUNC

' The last row of the log for an id; an empty map when it never ran.
EXPORT FUNC LAST(rows, id$)
    DIM found AS MAP
    DIM mine = FILTER(LAMBDA USE(id$) row -> row{"id"} = id$, rows)

```

```

    IF LEN(mine) = 0 THEN RETURN found
    RETURN mine[LEN(mine) - 1]
ENDFUNC

' How the last run ended, in words for the window.
EXPORT FUNC STATUS$(last)
    IF NOT MAP.EXISTS(last, "id") THEN RETURN "not run yet"
    DIM when$ = last{"time"}
    DIM how$ = "ok"
    IF last{"mode"} = "dry" THEN how$ = "dry run ok"
    IF last{"exit"} <> 0 THEN
        how$ = "failed (exit " + STR$(last{"exit"}) + ")"
        IF last{"mode"} = "dry" THEN how$ = "dry run " + how$
    ENDIF
    DIM s$ = how$ + ", " + RIGHT$(when$, 11)
    DIM msg$ = last{"message"}
    IF LEN(msg$) > 32 THEN msg$ = LEFT$(msg$, 30) + "..."
    IF last{"exit"} <> 0 AND ALSO msg$ <> "" THEN s$ = s$ + ": " + msg$
    RETURN s$
ENDFUNC

FUNC Recipe$(dir$, id$, name$, program$, code)
    MKDIR dir$
    DIM q$ = CHR$(34)
    DIM t$ = "id = " + q$ + id$ + q$ + gLF$
    t$ = t$ + "name = " + q$ + name$ + q$ + gLF$
    t$ = t$ + "program = " + q$ + program$ + q$ + gLF$
    TXTWRITER PATH.JOIN$(dir$, "recipe.toml"), t$
    TXTWRITER PATH.JOIN$(dir$, program$), JOIN(code, gLF$) + gLF$
    RETURN dir$
ENDFUNC

' A made-up work folder for trying the cockpit, in where$ or in a
' temporary folder: a Downloads Butler that only answers a dry run,
' a One-Click Backup that always fails, and the cockpit itself.
' Answers the folder; config/work.conf names jdbasic$.
EXPORT FUNC SAMPLE_WORK$(jdbasic$, where$)
    DIM work$ = where$
    IF work$ = "" THEN work$ = MKTEMP$("cockpit")
    MKDIR work$
    DIM q$ = CHR$(34)
    DIM root$ = PATH.JOIN$(work$, "recipes")
    DIM dry$ = q$ + "--dry-run" + q$
    DIM sorter = ["IF " + dry$ + " IN OS.ARGS() THEN", _
        "    PRINT " + q$ + "would sort 3 files" + q$, "ELSE", _
        "    THROW " + q$ + "only a dry run here" + q$, "ENDIF"]
    DIM fails = ["PRINT " + q$ + "copying" + q$, _
        "THROW " + q$ + "the share S: is not reachable" + q$]

```

```

DIM d$ = Recipe$(PATH.JOIN$(root$, "E01_butler"), "E01", _
    "Downloads Butler", "downloads_butler.jdb", sorter)
d$ = Recipe$(PATH.JOIN$(root$, "E04_one_click_backup"), "E04", _
    "One-Click Backup", "one_click_backup.jdb", fails)
d$ = Recipe$(PATH.JOIN$(root$, "X06_work_cockpit"), "X06", _
    "Work Cockpit", "work_cockpit.jdb", ["PRINT 0"])
MKDIR PATH.JOIN$(work$, "config")
DIM w$ = REPLACE$(work$, "\", "/")
DIM j$ = REPLACE$(jdbasic$, "\", "/")
DIM c$ = "[paths]" + gLF$ + "work = " + q$ + w$ + q$ + gLF$
c$ = c$ + "jdbasic = " + q$ + j$ + q$ + gLF$
TXTWRITER PATH.JOIN$(work$, "config", "work.conf"), c$
RETURN work$
ENDFUNC

```

How it works

1. **Finding the automations.** INSTALLED looks at each folder under recipes in the work folder and reads its `recipe.toml`. The id, the name and the program are all the cockpit needs. The cockpit's own folder, id X06, is left out, so it does not offer a button that opens a second cockpit.
1. **A button per automation.** The program creates the buttons in a loop and names them `btnRun0`, `btnRun1` and so on. A form calls a handler named after the control, such as `BTNRUN0_CLICK`, and `ON handler$ CALL RunClick` sends all of them to one handler. The event tells `RunClick` the name of the button that was clicked, and the number at its end is the place of the automation in the list.
1. **Starting an automation.** RUN starts `jdbasic` with the program, `--config` and the settings file, and `--dry-run` when the box is ticked. It waits until the program ends and keeps its exit code and output. With `tries` above 1, a run that fails is started again a second later, for automations that depend on a network drive that is not always there at once.
1. **The log.** `LOG_LINE$` writes one line per start, the fields separated by tabs, so a message with a comma or a semicolon stays in one field. `APPEND_LOG` adds the line at the end of the file. If another program holds the file open at that moment, it tries again three times before it gives up with an error.
1. **The state next to each button.** `READ_LOG` reads the log back as rows, and `LAST` finds the last row of each automation. `STATUS$` turns it into words

such as *dry run ok, 10-05 08:31* or *failed (exit 1), 10-05 09:15: the share S: is not reachable*, cut to fit the label. CLEAN\$ takes the colour codes out of the output, which jdBasic puts around its error messages for the console.

1. **The self test.** With `--selftest` the cockpit creates a made-up work folder with two automations, a Downloads Butler that answers a dry run and a backup that always fails, and drives its own window with a timer. It ticks the box, clicks both buttons by setting their `VALUE`, checks the labels and the log, and closes. No mouse and no keyboard are involved, so the test runs while you work.

Run it

Start the cockpit:

```
jdbasic work_cockpit.jdb
```

Tick *Dry run only* and click a button to see what an automation would do. Untick it to run it for real. *Refresh* reads the log again, which shows runs of other cockpit windows too.

The self test needs the jdbasic program, since it starts the made-up automations with it:

```
jdbasic work_cockpit.jdb --selftest --jdbasic C:/jdBasic/jdbasic.exe
ALL COCKPIT WINDOW TESTS PASSED
```

Schedule it

The cockpit is a window for a person, so it is not scheduled. To have it open every morning, set its schedule in the wizard to at logon. The automations themselves keep their own schedules, from the wizard or from the job server of X01; the cockpit is for the runs in between.

Make it yours

The setting is in the part of `work.conf` that starts with `[work_cockpit]`:

```
[work_cockpit]
tries = 1
```

- **More automations.** The window shows up to twelve. For more, raise the 12 in the line that sets gShown; the window grows by 36 pixels per row.
- **Only some of them.** To hide automations, filter the list after INSTALLED, for example only the ones whose id starts with E or M:

```
gList = FILTER(LAMBDA e -> LEFT$(e{"id"}, 1) <> "X", gList)
```

- **Show the log.** Add a button that puts the last 20 lines of the log into the output box:

```
SUB BTNLOG_CLICK(e)
  IF NOT FILE.EXISTS(gLog$) THEN RETURN
  DIM t$ = TRIM$(REPLACE$(TXTREADER$(gLog$), CHR$(13), ""))
  DIM tail = TAKE(-20, SPLIT(t$, LF$))
  FORM.SET(txtOut, "TEXT", JOIN(tail, CHR$(13) + LF$))
ENDSUB
```

and create the button next to *Refresh*:

```
DIM btnLog = FORM.BUTTON(frm, "btnLog", "Log", 370, top + 4, 90, 30)
```

When it goes wrong

- **No buttons, "Nothing installed.":** the work folder in [paths] is not the one the wizard used, or no recipe is installed yet. The first line of the window shows the folder the cockpit looks at.
- **Every run fails at once with exit 1:** the jdbasic program in [paths] is wrong or moved. The output box shows what Windows said, such as *The system cannot find the path specified*.
- **The window freezes for a while:** a run is in progress. The cockpit waits for the automation to end; long ones, such as a backup, keep the window busy until they are done.

Balance dividend

About 20 minutes a week of looking for folders, programs and log files, and an answer to *did the backup run?* in one glance.

X07 Tools for Colleagues

The chore

Lena's week sheet program has fans. Three colleagues want it, and none of them wants to install `jdBasic`, open a command prompt or learn what a module is. So far she copied the files to each of them by hand, and after every improvement she walked around with a USB stick. Twice a colleague kept an old version for weeks and sent her a sheet with a bug she had fixed long before. Copying, explaining and chasing old versions costs her close to an hour a week.

What you get

Three commands. `build` tests a recipe, compiles it into an `.exe` with everything it needs next to it, stamps it with a version and puts it into a release folder, for example on a shared drive. `check` tells a colleague whether a newer version is there. `update` brings their copy to the newest version, checks every file against its checksum before anything changes, and keeps the old version in a folder of its own.

```
tools_for_colleagues.exe check C:/Tools/week_sheet
week_sheet: installed 1.0.0, released 1.1.0
An update is ready.
```

The colleagues need nothing from `jdBasic`. The compiled program runs on any Windows computer, and the release folder may also be a web address inside the company.

Before you start

You need a folder that you may write to and your colleagues may read, such as `S:/Tools` on a shared drive, and the recipe you want to hand out, with its test. The compiler must be the `jdBasic` you work with, named in the `[paths]` part of `work.conf`, which the wizard wrote, and it must be a release with the native compiler: `jdbasic --version` lists *NativeC*. The forms release this book starts with has it, as Chapter 2 explains under *From Script to Program*.

A compiled recipe still reads its settings from a `work.conf`. Give your colleagues one with the part of the recipe, or start it with `--config` and a settings file on the shared drive.

Ask your IT department before you hand out programs. Many companies allow only programs from known publishers; a program you compiled yourself is not one of them until IT says so. The section *Talking to IT and Security* in Chapter 6 is about that conversation.

The program

The program reads the settings and the command, and calls the module:

```
' Tools for Colleagues: turns a recipe into a program colleagues can
' run without jdbBasic, puts each version into a release folder, and
' updates their copies from it. The settings are the
' [tools_for_colleagues] part of work.conf.
'
'   jdbbasic tools_for_colleagues.jdb build <recipe> --version 1.2.0
'   jdbbasic tools_for_colleagues.jdb check <tool folder>
'   jdbbasic tools_for_colleagues.jdb update <tool folder>
'   ... --dry-run                        show what would happen

IMPORT WORKCONF, SHIP

' The words on the command line that are not options.
FUNC Words(argv)
    DIM out = []
    DIM k = 1
    DO WHILE k < LEN(argv)
        DIM a$ = argv[k]
        IF a$ = "--version" ORELSE a$ = "--config" THEN
            k = k + 2
        ELSE
            IF NOT STARTSWITH(a$, "--") THEN PUSH(out, a$)
            k = k + 1
        ENDIF
    LOOP
    RETURN out
ENDFUNC

SUB WriteLog(file$, text$)
    DIM when$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M")
    TXTWRITER file$, when$ + " " + text$ + CHR$(10), TRUE
ENDSUB
```

```

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "tools_for_colleagues."
DIM rel_home$ = "~/Documents/AutomateWork/releases"
DIM release$ = WORKCONF.FOLDER$(cfg, key$ + "release", rel_home$)
DIM builds_home$ = "~/Documents/AutomateWork/builds"
DIM builds$ = WORKCONF.FOLDER$(cfg, key$ + "builds", builds_home$)
DIM jdbasic$ = WORKCONF.VALUE(cfg, "paths.jdbasic", "jdbasic")
DIM dry_run = WORKCONF.FLAG("dry-run")
DIM words = Words(OS.ARGS())
IF LEN(words) < 2 THEN THROW "Say build, check or update and a path."
DIM command$ = LCASE$(words[0])
DIM target$ = WORKCONF.EXPAND$(words[1])

IF command$ = "build" THEN
    DIM version$ = WORKCONF.ARG$("version")
    IF NOT SHIP.VALID_VERSION(version$) THEN THROW "Use --version"
    DIM name$ = SHIP.NAME$(target$)
    DIM src_dir$ = PATH.DIRNAME$(target$)
    DIM test$ = PATH.JOIN$(src_dir$, name$ + "_test.jdb")
    DIM stage$ = PATH.JOIN$(builds$, name$ + "-" + version$)
    DIM done_dir$ = PATH.JOIN$(release$, name$ + "-" + version$)
    IF FILE.ISDIR(done_dir$) THEN THROW version$ + " is released."
    IF dry_run THEN
        IF FILE.EXISTS(test$) THEN PRINT "would test with "; test$
        PRINT "would compile "; target$; " into "; stage$
        PRINT "would release "; name$; " "; version$; " to "; release$
    END
    ENDIF
    IF FILE.EXISTS(test$) THEN
        DIM t = OS.EXEC(jdbasic$, [test$])
        IF t{"EXIT_CODE"} <> 0 THEN THROW "The test fails."
        PRINT "test passed"
    ENDIF
    DIM b = SHIP.BUILD(jdbasic$, target$, stage$)
    IF NOT b{"ok"} THEN THROW "Compiling failed: " + b{"output"}
    SHIP.STAMP(stage$, name$, version$, NOW())
    DIM rel = SHIP.PACKAGE(stage$, release$, name$, version$)
    DIM done$ = "released " + name$ + " " + version$ + ", " + _
        STR$(rel{"files"}) + " files"
    PRINT done$; " in "; rel{"folder"}
    WriteLog(PATH.JOIN$(release$, "release.log"), done$)
ELSEIF command$ = "check" THEN
    DIM latest = SHIP.LATEST(release$)
    DIM have$ = SHIP.VERSION$(target$)
    IF have$ = "" THEN have$ = "none"
    PRINT latest{"name"}; ": installed "; have$; ", released "; _
        latest{"version"}
    IF SHIP.NEWER(SHIP.VERSION$(target$), latest{"version"}) THEN

```

```

        PRINT "An update is ready."
    ENDIF
ELSEIF command$ = "update" THEN
    DIM lines = SHIP.UPDATE(release$, target$, dry_run)
    DIM k = 0
    FOR k = 0 TO LEN(lines) - 1
        PRINT lines[k]
        DIM upd_log$ = PATH.JOIN$(target$, "update.log")
        IF NOT dry_run THEN WriteLog(upd_log$, lines[k])
    NEXT k
ELSE
    THROW "Unknown command " + command$ + "; use build, check, update"
ENDIF

```

The module compiles, releases and updates:

```

' ship.jdb: the work of Tools for Colleagues. A recipe is compiled
' into an .exe with everything it needs next to it, stamped with a
' version and put into a release folder with a checksum per file. A
' colleague's copy is brought up to date from that folder or from a
' web address.
'
'   IMPORT SHIP
'   DIM b = SHIP.BUILD(jdbasic$, "week_sheet.jdb", "C:/Builds/ws")
'   SHIP.STAMP(b{"stage"}, "week_sheet", "1.2.0", NOW())
'   DIM rel = SHIP.PACKAGE("C:/Builds/ws", "S:/Tools", "week_sheet", _
'       "1.2.0")
'   PRINT SHIP.UPDATE("S:/Tools", "C:/Tools/week_sheet", TRUE)

EXPORT MODULE SHIP

IMPORT CONF, PKG, RETRY

DIM gLF$ = CHR$(10)
DIM gQ$ = CHR$(34)

' The tool name of a program file: its name without .jdb.
EXPORT FUNC NAME$(program$)
    DIM base$ = PATH.BASENAME$(program$)
    IF LCASE$(RIGHT$(base$, 4)) = ".jdb" THEN
        base$ = LEFT$(base$, LEN(base$) - 4)
    ENDIF
    RETURN base$
ENDFUNC

' TRUE for a version of the form 1.2.0.
EXPORT FUNC VALID_VERSION(v$)

```

```

    RETURN LEN(REGEX.FINDALL("^([0-9]+\.[0-9]+\.[0-9]+)$", v$)) > 0
ENDFUNC

' Compiles program$ with jdbasic -c into the folder stage$. The
' compiler puts the runtime and its libraries next to the .exe.
' Answers "ok", "exe", "stage" and the compiler's text in "output".
EXPORT FUNC BUILD(jdbasic$, program$, stage$)
    DIM out AS MAP
    MKDIR stage$
    DIM exe$ = PATH.JOIN$(stage$, NAME$(program$) + ".exe")
    IF FILE.EXISTS(exe$) THEN KILL exe$
    DIM r = OS.EXEC(jdbasic$, ["-c", "-o", exe$, program$])
    DIM text$ = TRIM$(REPLACE$(r{"OUTPUT"}, CHR$(13), ""))
    out{"ok"} = r{"EXIT_CODE"} = 0 AND ALSO FILE.EXISTS(exe$)
    out{"exe"} = exe$
    out{"stage"} = stage$
    out{"output"} = text$
    RETURN out
ENDFUNC

' Writes version.toml into the folder: name, version and build time.
EXPORT SUB STAMP(stage$, name$, version$, built)
    DIM t$ = "name = " + gQ$ + name$ + gQ$ + gLF$
    t$ = t$ + "version = " + gQ$ + version$ + gQ$ + gLF$
    DIM when$ = FORMAT_DATE(built, "%Y-%m-%d %H:%M")
    t$ = t$ + "built = " + gQ$ + when$ + gQ$ + gLF$
    TXTWRITER PATH.JOIN$(stage$, "version.toml"), t$
ENDSUB

' The version in a folder's version.toml, "" when there is none.
EXPORT FUNC VERSION$(dir$)
    DIM file$ = PATH.JOIN$(dir$, "version.toml")
    IF NOT FILE.EXISTS(file$) THEN RETURN ""
    DIM t = CONF.TOML(file$)
    IF NOT MAP.EXISTS(t, "version") THEN RETURN ""
    DIM v$ = t{"version"}
    RETURN v$
ENDFUNC

' Copies every file of stage$ into release$/name-version, writes the
' list files.txt with a checksum per file, and points latest.toml at
' it. A version that is already released is an error. Answers
' "folder", "files" and the checksum of the list in "sha256".
EXPORT FUNC PACKAGE(stage$, release$, name$, version$)
    DIM dir_name$ = name$ + "-" + version$
    DIM dest$ = PATH.JOIN$(release$, dir_name$)
    IF FILE.ISDIR(dest$) THEN THROW dir_name$ + " is already released"
    MKDIR dest$

```



```

DIM names = SORT(DIR$(PATH.JOIN$(stage$, "*")))
DIM list = []
DIM k = 0
FOR k = 0 TO LEN(names) - 1
    DIM file_name$ = names[k]
    DIM path$ = PATH.JOIN$(stage$, file_name$)
    IF NOT FILE.ISDIR(path$) THEN
        FILE.COPY(path$, PATH.JOIN$(dest$, file_name$), FALSE)
        DIM file_sum$ = CODEC.SHA256$(BINREADER$(path$))
        PUSH(list, file_sum$ + " " + file_name$)
    ENDIF
NEXT k
DIM list$ = JOIN(list, gLF$) + gLF$
TXTWRITER PATH.JOIN$(dest$, "files.txt"), list$
DIM sum$ = CODEC.SHA256$(list$)
DIM t$ = "name = " + gQ$ + name$ + gQ$ + gLF$
t$ = t$ + "version = " + gQ$ + version$ + gQ$ + gLF$
t$ = t$ + "folder = " + gQ$ + dir_name$ + gQ$ + gLF$
t$ = t$ + "sha256 = " + gQ$ + sum$ + gQ$ + gLF$
DIM latest$ = PATH.JOIN$(release$, "latest.toml")
TXTWRITER latest$ + ".new", t$
FILE.MOVE(latest$ + ".new", latest$, TRUE)
DIM out AS MAP
out{"folder"} = dest$
out{"files"} = LEN(list)
out{"sha256"} = sum$
RETURN out
ENDFUNC

FUNC IsWeb(source$)
    DIM low$ = LCASE$(source$)
    IF STARTSWITH(low$, "http://") THEN RETURN TRUE
    RETURN STARTSWITH(low$, "https://")
ENDFUNC

FUNC GetUrl$(url$)
    DIM r = HTTP.REQUEST("GET", url$, "", "")
    DIM code = r{"status"}
    IF code <> 200 THEN THROW "HTTP " + STR$(code) + " for " + url$
    DIM body$ = r{"body"}
    RETURN body$
ENDFUNC

' A file of the release, from a folder or from a web address; file$
' may name a subfolder with a /. A web request is tried three times
' before it counts as failed.
EXPORT FUNC FETCH$(source$, file$)
    IF NOT IsWeb(source$) THEN

```

```

    DIM path$ = PATH.JOIN$(source$, file$)
    IF NOT FILE.EXISTS(path$) THEN THROW "Missing: " + file$
    RETURN BINREADER$(path$)
ENDIF
DIM url$ = source$
IF RIGHT$(url$, 1) <> "/" THEN url$ = url$ + "/"
url$ = url$ + REPLACE$(file$, " ", "%20")
DIM opts = {"attempts": 3, "delay": 300}
DIM report = RETRY.ATTEMPT(GetUrl$, url$, opts)
IF NOT report{"ok"} THEN THROW report{"error"}
DIM body$ = report{"value"}
RETURN body$
ENDFUNC

' The newest release as a map: name, version, folder and sha256.
EXPORT FUNC LATEST(source$)
    RETURN CONF.TOML_TEXT(FETCH$(source$, "latest.toml"))
ENDFUNC

' TRUE when latest$ is newer than installed$ ("" means not installed).
EXPORT FUNC NEWER(installed$, latest$)
    IF installed$ = "" THEN RETURN TRUE
    RETURN PKG.COMPARE(installed$, latest$) < 0
ENDFUNC

' Every file of a release, checked against its list, into a new
' temporary folder. Answers the folder; a wrong checksum is an error.
FUNC Download$(source$, latest$)
    DIM dir$ = latest{"folder"}
    DIM list$ = FETCH$(source$, dir$ + "/files.txt")
    IF CODEC.SHA256$(list$) <> latest{"sha256"} THEN
        THROW dir$ + ": the checksum of files.txt does not match"
    ENDIF
    DIM temp$ = MKTEMP$("ship")
    MKDIR temp$
    DIM rows = SPLIT(TRIM$(REPLACE$(list$, CHR$(13), "")), gLF$)
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM row$ = rows[k]
        DIM sum$ = LEFT$(row$, 64)
        DIM file_name$ = MID$(row$, 66, LEN(row$))
        DIM data$ = FETCH$(source$, dir$ + "/" + file_name$)
        IF CODEC.SHA256$(data$) <> sum$ THEN
            OS.EXEC("cmd", ["/c", "rmdir", "/s", "/q", temp$])
            THROW file_name$ + ": the checksum does not match"
        ENDIF
        BINWRITER PATH.JOIN$(temp$, file_name$), data$
    NEXT k

```

```

    RETURN temp$
ENDFUNC

' Brings the tool in target$ to the newest release of source$. Every
' file is downloaded and checked before anything changes; the old
' files go into target$/previous. With dry_run it only says what it
' would do. Answers the lines it reports.
EXPORT FUNC UPDATE(source$, target$, dry_run)
    DIM lines = []
    DIM latest = LATEST(source$)
    DIM have$ = VERSION$(target$)
    DIM want$ = latest{"version"}
    DIM shown$ = have$
    IF shown$ = "" THEN shown$ = "nothing"
    IF NOT NEWER(have$, want$) THEN
        PUSH(lines, latest{"name"} + " " + have$ + " is up to date")
        RETURN lines
    ENDIF
    IF dry_run THEN
        DIM plan$ = "would update " + latest{"name"}
        PUSH(lines, plan$ + " from " + shown$ + " to " + want$)
        RETURN lines
    ENDIF
    DIM temp$ = Download$(source$, latest)
    MKDIR target$
    DIM keep$ = PATH.JOIN$(target$, "previous")
    MKDIR keep$
    DIM old = DIR$(PATH.JOIN$(target$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(old) - 1
        DIM old_name$ = old[k]
        DIM old_path$ = PATH.JOIN$(target$, old_name$)
        IF NOT FILE.ISDIR(old_path$) THEN
            FILE.COPY(old_path$, PATH.JOIN$(keep$, old_name$), TRUE)
        ENDIF
    NEXT k
    DIM names = DIR$(PATH.JOIN$(temp$, "*"))
    FOR k = 0 TO LEN(names) - 1
        DIM new_name$ = names[k]
        DIM new_path$ = PATH.JOIN$(temp$, new_name$)
        FILE.MOVE(new_path$, PATH.JOIN$(target$, new_name$), TRUE)
    NEXT k
    OS.EXEC("cmd", ["/c", "rmdir", "/s", "/q", temp$])
    DIM done$ = "updated " + latest{"name"} + " from " + shown$
    DIM count$ = STR$(LEN(names)) + " files"
    PUSH(lines, done$ + " to " + want$ + ", " + count$)
    RETURN lines
ENDFUNC

```

How it works

1. **Test before release.** `build` looks for the recipe's test next to it, `week_sheet_test.jdb` for `week_sheet.jdb`, and runs it with `jdBasic`. When the test fails, nothing is compiled and nothing is released. A version that is already in the release folder is refused as well, so a released version never changes under the feet of a colleague who has it.
1. **Compiling.** `BUILD` runs `jdbasic -c -o` with the path of the `.exe`. The compiler puts the runtime `jdbrt.dll`, the libraries it needs and the default font next to the program. That folder runs on its own, on a computer without `jdBasic`. A program that does not compile gives ok as `FALSE` and the compiler's message in output.
1. **The version.** `STAMP` writes `version.toml` into the folder, with the name, the version and the time of the build. `VERSION$` reads it back, on your computer and on your colleagues'. Versions are three numbers such as `1.2.0`. `NEWER` compares them with `PKG.COMPARE` from the package library, which knows that `1.10.0` comes after `1.9.0`.
1. **The release.** `PACKAGE` copies the folder into the release folder as `week_sheet-1.2.0` and writes `files.txt`, one line per file with its SHA-256 checksum. Then it writes `latest.toml`, which names the newest version and the checksum of `files.txt`. It writes it under another name first and moves it into place, so a colleague who checks at that moment reads the old file or the new one, never half of it.
1. **Fetching.** `FETCH$` reads a file of the release from a folder, or from a web address when the release folder starts with `http`. A request that fails is tried three times with a growing pause by the `RETRY` library, since a web server that is busy for a second answers the next time.
1. **Checking every byte.** `Download$` fetches `files.txt` and compares its checksum with the one in `latest.toml`, then fetches each file and compares its checksum with its line. Everything goes into a temporary folder first. A single wrong checksum stops the update before the colleague's copy is touched.
1. **The update.** `UPDATE` copies the old files into `previous` and moves the new ones in. A colleague whose new version misbehaves copies the files of `previous` back and has the old one again.

Run it

Release version 1.0.0 of the week sheet. Add `--dry-run` first to see what it would test, compile and release. The real run:

```
jdbasic tools_for_colleagues.jdb build week_sheet.jdb --version 1.0.0
test passed
released week_sheet 1.0.0, 10 files in S:/Tools\week_sheet-1.0.0
```

The same version a second time is refused:

```
jdbasic tools_for_colleagues.jdb build week_sheet.jdb --version 1.0.0
Error #99: 1.0.0 is released. at line 55
```

On the colleague's computer, the compiled tool checks and updates:

```
tools_for_colleagues.exe check C:/Tools/week_sheet
week_sheet: installed none, released 1.0.0
An update is ready.
tools_for_colleagues.exe update C:/Tools/week_sheet
updated week_sheet from nothing to 1.0.0, 10 files
```

After you released 1.1.0, the next update says:

```
updated week_sheet from 1.0.0 to 1.1.0, 10 files
```

Each version takes about 20 MB in the release folder, most of it the runtime that travels with every version.

Schedule it

Releasing is something you decide, so `build` runs by hand. The update can run on its own on each colleague's computer: a task in the Task Scheduler at logon, or a job of the X01 server on yours for the copy you use yourself:

```
[[job]]
name = "update week sheet"
when = "weekdays 08:00"
program = "X07_tools_for_colleagues/tools_for_colleagues.jdb"
args = ["update", "C:/Tools/week_sheet"]
```

Make it yours

The settings are in the part of `work.conf` that starts with `[tools_for_colleagues]:`

```
[tools_for_colleagues]
release = "S:/Tools"
builds = "~/Documents/AutomateWork/builds"
```

- **A web address.** Put the release folder on an internal web server and set `release = "http://tools.example.local/tools"`. Check and update work the same way; only build needs the folder itself, so release with a second settings file that names the folder.
- **More than one tool.** One release folder holds one tool, because `latest.toml` names one version. Give each tool its own folder, such as `S:/Tools/week_sheet` and `S:/Tools/invoices`.
- **Clean up old versions.** Each release stays in the folder. Delete folders of versions nobody uses any more by hand; `latest.toml` points only to the newest one.
- **A shortcut for colleagues.** Put a link to the `.exe` on their desktop, with `--config` and the settings file in the link's target, so a double click starts the tool.

When it goes wrong

- **"The test fails":** the recipe's test does not pass. Run it yourself with `jdbasic week_sheet_test.jdb` and fix what it shows.
- **"Compiling failed":** the program uses something the compiler rejects, such as a variable without `DIM`. The message names the line. The interpreter is more forgiving, which is why a program can run there and still fail to compile.
- **"the checksum does not match":** a file of the release changed after it was released, or the download was cut off. Release a new version; never change a released folder.
- **The program does not start on a colleague's computer:** Windows or the virus scanner blocked an unknown program. That is the conversation with IT from *Before you start*.

Balance dividend

About 45 minutes a week of copying, explaining and chasing old versions, and colleagues who always run the version you fixed.

X08 Parallel Crunching

The chore

At the start of every month the travel booking tool hands Lena the expenses of the month before: one CSV file per cost centre and day, about two thousand files. Finance wants one table out of them, the amount per category for hotels, meals, taxis and trains. Her first program for it read the files one after the other and took several minutes, during which her laptop used one of its eight cores and left the other seven alone. Worse, when one file in the folder was not an export at all, the program stopped halfway and she started over.

What you get

A program that hands the files to as many workers as the computer has cores, adds their answers into one report, and writes it as a CSV file next to the earlier ones:

```
category;amount
hotel;450700.00
meals;450610.00
taxi;450700.00
train;450880.00
total;1802890.00
```

A file that is no export does not stop the run. It is named in the summary and in the log, and the other files are added up. Lines in an export that cannot be read, such as an amount written as a word, are counted and skipped.

Before you start

Put the exports of one month into a folder of their own, one folder per month, such as exports/2026-09. Each file needs a header line with the columns date, category and amount, separated by the separator of the settings. The recipe reads the files and never changes them.

For the speed this page describes, compile the program first, as X07 shows. The interpreter runs the same program with the same result, but there the workers gain little: the run takes about as long as on one core.

The program

The program reads the settings, lets the module do the work, and writes the report and a log line:

```
' Parallel Crunching: adds up a folder of expense exports on every
' core of the computer and writes one report with the amount per
' category. Files that cannot be read are listed, not guessed. The
' settings are the [parallel_crunching] part of work.conf.
'
'   jdbasic parallel_crunching.jdb           crunch the folder
'   jdbasic parallel_crunching.jdb --folder <dir>  another folder
'   jdbasic parallel_crunching.jdb --serial        on one core
'   jdbasic parallel_crunching.jdb --dry-run       count only

IMPORT WORKCONF, CRUNCH

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "parallel_crunching."
DIM home$ = "~/Documents/AutomateWork/exports"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM given$ = WORKCONF.ARG$("folder")
IF given$ <> "" THEN folder$ = WORKCONF.EXPAND$(given$)
DIM reports_home$ = "~/Documents/AutomateWork/reports"
DIM out$ = WORKCONF.FOLDER$(cfg, key$ + "reports", reports_home$)
DIM sep$ = WORKCONF.VALUE(cfg, key$ + "separator", ";")
DIM wanted = WORKCONF.VALUE(cfg, key$ + "workers", 0)
DIM limit = WORKCONF.VALUE(cfg, key$ + "timeout_seconds", 30)
DIM workers = CRUNCH.WORKERS(CINT(wanted))

DIM files = CRUNCH.FILES(folder$)
IF LEN(files) = 0 THEN
    PRINT "No CSV files in "; folder$
    END
ENDIF
IF WORKCONF.FLAG("dry-run") THEN
    PRINT "would crunch "; LEN(files); " files in "; folder$
    PRINT "on "; workers; " workers, report into "; out$
    END
ENDIF

DIM run AS MAP
IF WORKCONF.FLAG("serial") THEN
    run = CRUNCH.SERIAL(files, sep$)
    workers = 1
ELSE
    run = CRUNCH.PARALLEL(files, sep$, workers, limit * 1000)
ENDIF
ENDIF
```

```

DIM texts = run{"texts"}
DIM failed = run{"failed"}
DIM ms = run{"ms"}
DIM totals = CRUNCH.MERGE(texts)

MKDIR out$
DIM base$ = "expenses " + PATH.BASENAME$(folder$)
DIM report$ = PATH.JOIN$(out$, base$ + ".csv")
DIM n = 2
DO WHILE FILE.EXISTS(report$)
    report$ = PATH.JOIN$(out$, base$ + " " + STR$(n) + ".csv")
    n = n + 1
LOOP
TXTWRITER report$, CRUNCH.REPORT_CSV$(totals, sep$)

DIM lines = []
PUSH(lines, FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M") + " " + folder$)
PUSH(lines, " " + STR$(totals{"files"}) + " files, " + _
    STR$(totals{"rows"}) + " rows, " + STR$(totals{"bad"}) + _
    " lines skipped, " + STR$(workers) + " workers, " + _
    STR$(INT(ms)) + " ms")
DIM total$ = CRUNCH.MONEY$(totals{"cents"})
PUSH(lines, " total " + total$ + ", report " + report$)
DIM not_read = SELECT(LAMBDA s$ -> " not read: " + s$, failed)
lines = APPEND(lines, not_read)
DIM text$ = JOIN(lines, CHR$(10)) + CHR$(10)
PRINT text$;
TXTWRITER PATH.JOIN$(out$, "parallel_crunching.log"), text$, TRUE
IF LEN(failed) > 0 THEN THROW STR$(LEN(failed)) + " files not read"

```

The module reads one export, runs all of them on one core or on many, and adds the answers up:

```

' crunch.jdb: the work of Parallel Crunching. Each expense export
' becomes one line of text (rows, amount, unreadable lines, amount
' per category), on one core or on many, and the lines are added up
' into one report.
'
'   IMPORT CRUNCH
'   DIM files = CRUNCH.FILES("C:/Exports/2026-09")
'   DIM run = CRUNCH.PARALLEL(files, ";", 8, 30000)
'   DIM totals = CRUNCH.MERGE(run{"texts"})

EXPORT MODULE CRUNCH

IMPORT POOL

```

```

' The CSV files of a folder, sorted, as full paths.
EXPORT FUNC FILES(folder$)
    DIM out = []
    IF NOT FILE.ISDIR(folder$) THEN RETURN out
    DIM names = SORT(DIR$(PATH.JOIN$(folder$, "*.csv")))
    DIM paths = SELECT(LAMBDA USE(folder$) n$ -> _
        PATH.JOIN$(folder$, n$), names)
    RETURN FILTER(LAMBDA p$ -> NOT FILE.ISDIR(p$), paths)
ENDFUNC

' One export as text: "rows;cents;bad;cat=cents,cat=cents". The file
' starts with the header date, category, amount; another file is an
' error.
EXPORT FUNC SUMMARY$(path$, sep$)
    DIM text$ = REPLACE$(TXTREADER$(path$), CHR$(13), "")
    DIM lines = SPLIT(text$, CHR$(10))
    DIM head$ = LCASE$(REPLACE$(lines[0], " ", ""))
    IF head$ <> "date" + sep$ + "category" + sep$ + "amount" THEN
        THROW "not an expense export"
    ENDIF
    DIM by AS MAP
    DIM rows = 0
    DIM cents = 0
    DIM bad = 0
    DIM k = 0
    FOR k = 1 TO LEN(lines) - 1
        DIM line$ = TRIM$(lines[k])
        IF line$ <> "" THEN
            DIM parts = SPLIT(line$, sep$)
            DIM ok = LEN(parts) = 3
            DIM num_rx$ = "^-?[0-9]+(\\.[0-9]+)?$"
            IF ok THEN ok = LEN(REGEX.FINDALL(num_rx$, parts[2])) = 1
            IF ok THEN
                DIM cat$ = TRIM$(parts[1])
                DIM c = CINT(ROUND(VAL(parts[2]) * 100))
                rows = rows + 1
                cents = cents + c
                IF MAP.EXISTS(by, cat$) THEN
                    by{cat$} = by{cat$} + c
                ELSE
                    by{cat$} = c
                ENDIF
            ELSE
                bad = bad + 1
            ENDIF
        ENDIF
    NEXT k
    DIM cats = SORT(MAP.KEYS(by))

```

```

    DIM pairs = []
    FOR k = 0 TO LEN(cats) - 1
        DIM name$ = cats[k]
        PUSH(pairs, name$ + "=" + STR$(by{name$}))
    NEXT k
    DIM nums$ = STR$(rows) + ";" + STR$(cents) + ";" + STR$(bad)
    RETURN nums$ + ";" + JOIN(pairs, ",")
ENDFUNC

' Every file on this core. Answers the same map as PARALLEL.
EXPORT FUNC SERIAL(files, sep$)
    DIM started = TICK()
    DIM texts = []
    DIM failed = []
    DIM k = 0
    FOR k = 0 TO LEN(files) - 1
        DIM p$ = files[k]
        TRY
            PUSH(texts, SUMMARY$(p$, sep$))
        CATCH
            PUSH(texts, "")
            PUSH(failed, PATH.BASENAME$(p$) + ": " + ERRMSG$)
        ENDTRY
    NEXT k
    DIM out AS MAP
    out{"texts"} = texts
    out{"failed"} = failed
    out{"ms"} = TICK() - started
    RETURN out
ENDFUNC

ASYNC FUNC Worker(jobs, results, files, sep$)
    DO
        DIM idx = POOL.TAKE_JOB(jobs, results)
        IF idx < 0 THEN EXITDO
        TRY
            DIM p$ = files[idx]
            POOL.DONE_TEXT(results, idx, SUMMARY$(p$, sep$))
        CATCH
            POOL.FAILED(results, idx, ERRMSG$)
        ENDTRY
    LOOP
    RETURN 0
ENDFUNC

' Every file on several workers. Answers "texts" in input order
' (" for a file that failed), "failed" as lines "name: reason",
' and "ms".

```

```

EXPORT FUNC PARALLEL(files, sep$, workers, timeout_ms)
    DIM started = TICK()
    DIM out AS MAP
    out{"texts"} = []
    out{"failed"} = []
    IF LEN(files) = 0 THEN
        out{"ms"} = 0
        RETURN out
    ENDIF
    DIM p = POOL.CREATE(LEN(files))
    DIM w = 0
    FOR w = 1 TO MAX([1, workers])
        DIM task = Worker(POOL.JOBS(p), POOL.RESULTS(p), files, sep$)
        POOL.SPAWN(p, task)
    NEXT w
    DIM good = POOL.GATHER(p, timeout_ms, FALSE)
    DIM texts = []
    DIM failed = []
    DIM k = 0
    FOR k = 0 TO LEN(files) - 1
        IF POOL.STATE$(p, k) = "ok" THEN
            PUSH(texts, POOL.VALUE_TEXT$(p, k))
        ELSE
            PUSH(texts, "")
            DIM file$ = files[k]
            DIM why$ = POOL.REASON$(p, k)
            PUSH(failed, PATH.BASENAME$(file$) + ": " + why$)
        ENDIF
    NEXT k
    POOL.SHUTDOWN(p)
    out{"texts"} = texts
    out{"failed"} = failed
    out{"ms"} = TICK() - started
    RETURN out
ENDFUNC

```

' The texts added up: "files", "rows", "cents", "bad" and "by", the
' cents per category. A text that is "" counts as a file left out.

```

EXPORT FUNC MERGE(texts)
    DIM by AS MAP
    DIM files = 0
    DIM rows = 0
    DIM cents = 0
    DIM bad = 0
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(texts) - 1
        DIM t$ = texts[k]

```

```

    IF t$ <> "" THEN
        DIM parts = SPLIT(t$, ";")
        files = files + 1
        rows = rows + CINT(VAL(parts[0]))
        cents = cents + CINT(VAL(parts[1]))
        bad = bad + CINT(VAL(parts[2]))
        IF parts[3] <> "" THEN
            DIM pairs = SPLIT(parts[3], ",")
            FOR j = 0 TO LEN(pairs) - 1
                DIM pair = SPLIT(pairs[j], "=")
                DIM cat$ = pair[0]
                DIM c = CINT(VAL(pair[1]))
                IF MAP.EXISTS(by, cat$) THEN
                    by{cat$} = by{cat$} + c
                ELSE
                    by{cat$} = c
                ENDIF
            NEXT j
        ENDIF
    ENDIF
NEXT k
DIM out AS MAP
out{"files"} = files
out{"rows"} = rows
out{"cents"} = cents
out{"bad"} = bad
out{"by"} = by
RETURN out
ENDFUNC

' An amount in cents as text with two decimals.
EXPORT FUNC MONEY$(cents)
    DIM sign$ = ""
    DIM c = cents
    IF c < 0 THEN
        sign$ = "-"
        c = -c
    ENDIF
    DIM whole = INT(c / 100)
    DIM rest = c - whole * 100
    DIM rest$ = STR$(rest)
    IF rest < 10 THEN rest$ = "0" + rest$
    RETURN sign$ + STR$(whole) + "." + rest$
ENDFUNC

' The report as CSV: one line per category and a total.
EXPORT FUNC REPORT_CSV$(totals, sep$)
    DIM by = totals{"by"}

```

```

    DIM cats = SORT(MAP.KEYS(by))
    DIM rows = SELECT(LAMBDA USE(by, sep$) n$ -> _
        n$ + sep$ + MONEY$(by{n$}), cats)
    DIM lines = APPEND(["category" + sep$ + "amount"], rows)
    PUSH(lines, "total" + sep$ + MONEY$(totals{"cents"}))
    RETURN JOIN(lines, CHR$(10)) + CHR$(10)
ENDFUNC

' The number of workers for a setting: 0 means one per core.
EXPORT FUNC WORKERS(setting)
    IF setting > 0 THEN RETURN setting
    DIM n = CINT(VAL(GETENV$("NUMBER_OF_PROCESSORS")))
    IF n < 1 THEN n = 4
    RETURN n
ENDFUNC

```

How it works

1. **A line of text per file.** SUMMARY\$ reads an export and answers a short line such as 20;63500;0;hotel=31750,taxi=31750: the number of rows, the total in cents, the lines it skipped, and the cents per category. Everything a worker hands back is in that line, so the workers share nothing while they work.
1. **Cents, not euros.** Amounts are added as whole cents. Adding decimal numbers in a different order can change the last digit, and the order in which workers finish is never the same twice. With cents the parallel report is always identical to the serial one, and the test checks exactly that.
1. **The worker.** Worker is an ASYNC FUNC: each call runs on a thread of its own. It takes the next job number from the pool with POOL.TAKE_JOB, reads that file, and reports the line with POOL.DONE_TEXT. When a file throws an error, POOL.FAILED hands the message back instead, and the worker goes on with the next job. When no job is left, TAKE_JOB answers -1 and the worker ends.
1. **The pool.** PARALLEL creates a pool with one job per file and starts as many workers as asked for. POOL.GATHER waits until every job has ended, and a job that takes longer than the timeout counts as failed. The answers come back in the order of the files, no matter which worker finished first.
1. **One core for comparison.** SERIAL does the same work in a plain loop and answers the same map. Run the program with --serial when you want to

see what the workers gain, or when you look for a problem and want one thing to happen at a time.

1. **Adding up.** MERGE splits the lines again and adds rows, cents and skipped lines, and the cents of each category. REPORT_CSV\$ turns the totals into the report, and MONEY\$ writes cents with two decimals.
1. **No report is overwritten.** When expenses 2026-09.csv exists already, the next run writes expenses 2026-09 2.csv, so a second run after a correction keeps the first result for comparison.

Run it

A dry run counts the files and says where the report would go:

```
parallel_crunching.exe --dry-run
would crunch 2001 files in C:/Exports/2026-09
on 8 workers, report into C:/Reports
```

The real run prints the summary that also goes into the log. Here one file of the folder, readme.csv, is a note and no export:

```
parallel_crunching.exe
2026-10-05 08:12 C:/Exports/2026-09
  2000 files, 40000 rows, 5 lines skipped, 8 workers, 338 ms
total 1802890.00, report C:/Reports\expenses 2026-09.csv
not read: readme.csv: not an expense export
```

A run in which a file was not read ends with an error, so the job server of X01 and the cockpit of X06 show it as failed, although the report was written.

How much the workers gain depends on the work per file. For two thousand exports of 300 lines each, on a notebook with eight threads, the compiled program needed 4.2 seconds on one core and 1.5 seconds with eight workers. The interpreter needed about six seconds either way.

Schedule it

The exports arrive once a month, so the wizard leaves the recipe to be run by hand. With the job server of X01 it runs on the second working day of the month, after the export has arrived:


```
[[job]]
name = "expenses"
when = "0 9 2 * *"
program = "X08_parallel_crunching/parallel_crunching.jdb"
args = ["--folder", "C:/Exports/latest"]
```

Make it yours

The settings are in the part of `work.conf` that starts with

`[parallel_crunching]:`

```
[parallel_crunching]
folder = "C:/Exports/2026-09"
reports = "~/Documents/AutomateWork/reports"
separator = ";"
workers = 0
timeout_seconds = 30
```

`workers = 0` means one worker per core. On a computer that has other work to do while the recipe runs, give it fewer.

- **Other columns.** If your exports have the amount in the fifth column and the category in the third, change the two indexes in `SUMMARY$` and the check `LEN(parts) = 3` to the number of columns.
- **Totals per cost centre.** Put the file name in front of the category in `SUMMARY$`, so a key reads `cc0412.csv/hotel`:

```
DIM cat$ = PATH.BASENAME$(path$) + "/" + TRIM$(parts[1])
```

`REPORT_CSV$` sorts the keys, so the report groups by cost centre.

- **Other work per file.** Anything that turns one file into one line of text fits the same pool: the page count of PDF files, the words of a set of Word documents, the checksum of every file of a backup. Write a new `SUMMARY$` and a `MERGE` that adds its lines.

When it goes wrong

- **"No CSV files in":** the folder is wrong, or the exports end in .txt. Change the pattern in FILES.
- **"not an expense export"** for every file: the header uses another separator or other names. Look at the first line of one file and set separator, or change the header check in SUMMARY\$.
- **Many skipped lines:** the amounts use a decimal comma, such as 12,50. Replace the comma with `REPLACE$(parts[2], ",", ".")` before the check.
- **A file timed out:** it is very large or lies on a slow network drive. Raise `timeout_seconds`, or copy the folder to the local disk first.

Balance dividend

About 40 minutes a month, ten a week, of waiting and starting over, and a report that names every file it left out instead of a total nobody can check.

X09 Report Pipeline

The chore

Every Monday at nine the head of Lena's department wants to know what the team spent last week, per cost center, against the budget. The numbers are in the bookings database that the finance system fills every night. Lena opens it in Excel, filters the week, builds a pivot table, copies it into last week's Word file, draws the bars for the share of the budget by hand, saves a PDF and writes the mail. On a good Monday that is 45 minutes before her own work starts. On a bad one the export is still running at eight, the numbers are from Saturday, and nobody notices until the meeting.

What you get

Every Monday at seven the pipeline reads last week's bookings from the database and writes three files:

- a chart of spent and budget per cost center as SVG, the picture format every browser and the intranet show,
- a Word file with the table and a list of the cost centers over budget, for anyone who wants to add a comment,
- a PDF page with the table and a bar per cost center that shows how much of its budget is used, red when it is over.

It puts the mail with the PDF and the Word file attached into the outbox, and every step leaves a line in a log.

Team costs

Week of 28 September 2026, 2026-09-28 to 2026-10-04

Cost center	Spent	Budget	Used
Sales	2,920.00	4,000.00	73 %
IT	2,010.00	3,000.00	67 %
Marketing	1,750.00	2,500.00	70 %
Travel	876.00	1,200.00	73 %
Total	7,556.00	10,700.00	

Share of the budget used



The PDF the pipeline writes for the week of 28 September

Before you start

The recipe reads a SQLite database, the kind of single file database many programs use to keep their data, with two tables:

Table	Columns
bookings	day (YYYY-MM-DD), cost_center, amount
budgets	cost_center, week_budget

If your finance system exports a CSV file instead, the Hot Folder of M10 or a short loop with `DB.INSERTMANY` fills the table; the section *Make it yours* shows the lines. To try the pipeline before you have a database, let it write a sample with eight weeks of bookings:

```
jdbasic report_pipeline.jdb --sample
```

The recipe's part of `work.conf` names the database, the folder for the reports and who gets the mail:

```
[report_pipeline]
database = "~/Documents/AutomateWork/reports/bookings.db"
folder = "~/Documents/AutomateWork/reports"
to = "Paul Berger <paul.berger@example.com>"
```

The program

The program is a pipeline in the plain sense: each step takes what the step before handed it. It reads the settings, finds the week, reads the database, makes the report and writes it out:

```
' Report Pipeline: every Monday morning last week's team costs from
' the bookings database, per cost center against the budget, as a
' chart, a Word file and a PDF, with the mail that carries them in the
' outbox. The settings are the [report_pipeline] part of work.conf.
'
'   jdbasic report_pipeline.jdb                last week
'   jdbasic report_pipeline.jdb --day 2026-10-12  as on that day
'   jdbasic report_pipeline.jdb --dry-run         show, write nothing
'   jdbasic report_pipeline.jdb --sample         a sample database
'   jdbasic report_pipeline.jdb --send           send the outbox

IMPORT WORKCONF, PIPELINE, OUTBOX, LOGGER, SVG

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
    PRINT result{"sent"}; " sent"
    DIM f = 0
    FOR f = 0 TO LEN(result{"failed"}) - 1
        PRINT "  not sent: "; result{"failed"}[f]
    NEXT f
    END
ENDIF

DIM key$ = "report_pipeline."
DIM home$ = "~/Documents/AutomateWork/reports"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM db_home$ = PATH.JOIN$(folder$, "bookings.db")
DIM db$ = WORKCONF.FOLDER$(cfg, key$ + "database", db_home$)
DIM to$ = WORKCONF.VALUE(cfg, key$ + "to", "")
DIM day$ = WORKCONF.ARG$("day")
IF day$ = "" THEN day$ = FORMAT_DATE(NOW(), "%Y-%m-%d")

MKDIR folder$
DIM lg = LOGGER.NEW("report_pipeline")
```

```

LOGGER.TO_FILE(lg, PATH.JOIN$(folder$, "report_pipeline.log"), _
    1000000, 3)

IF WORKCONF.FLAG("sample") THEN
    IF FILE.EXISTS(db$) THEN THROW "There is a database at " + db$
    PIPELINE.SAMPLE(db$, day$)
    LOGGER.INFO(lg, "sample database written", {"path": db$})
    END
ENDIF

DIM week = PIPELINE.LAST_WEEK(day$)
LOGGER.INFO(lg, "report started", {"from": week{"from"}})
DIM rows = []
DIM budgets AS MAP
TRY
    rows = PIPELINE.BOOKINGS(db$, week)
    budgets = PIPELINE.BUDGETS(db$)
CATCH
    LOGGER.ERROR(lg, "database failed", {"error": ERRMSG$})
    THROW "Report Pipeline: " + ERRMSG$
ENDTRY
IF LEN(rows) = 0 THEN LOGGER.WARN(lg, "no bookings in the week")

DIM report = PIPELINE.REPORT(rows, budgets)
DIM table = PIPELINE.TABLE_ROWS(report)
DIM over = PIPELINE.OVER_BUDGET(report)
DIM base$ = PATH.JOIN$(folder$, "costs " + week{"from"})

IF WORKCONF.FLAG("dry-run") THEN
    PRINT week{"label"}; ": "; LEN(rows); " bookings"
    DIM r = 0
    FOR r = 1 TO LEN(table) - 1
        DIM row = table[r]
        DIM cc$ = row[0]
        DIM spent$ = row[1]
        DIM budget$ = row[2]
        PRINT " "; cc$; ": "; spent$; " of "; budget$
    NEXT r
    DIM o = 0
    FOR o = 0 TO LEN(over) - 1
        DIM over$ = over[o]
        PRINT " over budget: "; over$
    NEXT o
    PRINT "would write "; base$; ".svg, .docx and .pdf"
    IF to$ <> "" THEN PRINT "would put a mail to "; to$
    END
ENDIF

```

```

TXTWRITER base$ + ".svg", PIPELINE.CHART$(report, week)
PIPELINE.WRITE_DOCX(report, week, base$ + ".docx")
PIPELINE.WRITE_PDF(report, week, base$ + ".pdf")
LOGGER.INFO(lg, "report written", {"base": base$, "over": LEN(over)})
PRINT "Wrote "; base$; ".svg, .docx and .pdf"
IF to$ = "" THEN
    PRINT "No [report_pipeline] to in work.conf, so no mail."
ELSE
    DIM msg = PIPELINE.MESSAGE(OUTBOX.FROM$(cfg), to$, report, week, _
        base$ + ".pdf", base$ + ".docx")
    DIM eml$ = OUTBOX.PUT$(msg, cfg, "costs " + week{"from"} + ".eml")
    LOGGER.INFO(lg, "mail in the outbox", {"path": eml$})
    PRINT "In the outbox: "; eml$
ENDIF

```

The module holds every step as a function of its own:

```

' pipeline.jdb: the weekly cost report of a team. Bookings come from a
' SQLite database, a data frame adds them up per cost center against
' the budget, and the result is written as a chart (SVG), a Word file
' and a PDF, and put into a mail.
'
'   IMPORT PIPELINE
'   DIM week = PIPELINE.LAST_WEEK("2026-10-05")
'   DIM rows = PIPELINE.BOOKINGS(db$, week)
'   DIM report = PIPELINE.REPORT(rows, PIPELINE.BUDGETS(db$))
'   PIPELINE.WRITE_PDF(report, week, "costs.pdf")

EXPORT MODULE PIPELINE

IMPORT DB, DF, SVG, DOCX, PDFGEN, MAIL, RETRY

' The Monday to Sunday before the week of day$ (YYYY-MM-DD), as a map
' with "from", "to" and "label". The days are counted from noon, so
' the hour that daylight saving time adds or takes keeps the date.
EXPORT FUNC LAST_WEEK(day$)
    DIM d = CDATE(day$ + " 12:00:00")
    DIM back = (VAL(FORMAT_DATE(d, "%w")) + 6) MOD 7
    DIM monday = DATEADD("D", -back - 7, d)
    DIM from$ = FORMAT_DATE(monday, "%Y-%m-%d")
    DIM to$ = FORMAT_DATE(DATEADD("D", 6, monday), "%Y-%m-%d")
    DIM label$ = "Week of " + FORMAT_DATE(monday, "%d %B %Y")
    RETURN {"from": from$, "to": to$, "label": label$}
ENDFUNC

FUNC OpenDb(path$)
    IF NOT FILE.EXISTS(path$) THEN THROW "No database at " + path$

```

```

    RETURN DB.OPEN(path$)
ENDFUNC

' The bookings of the week as rows with "day", "cost_center" and
' "amount". Opening the database is tried up to four times, with a
' growing wait between the attempts.
EXPORT FUNC BOOKINGS(db_path$, week)
    DIM opts = {"attempts": 4, "delay": 250}
    DIM h = RETRY.RUN(OpenDb@, db_path$, opts)
    DIM q = DB.FROM("bookings")
    DB.WHERE(q, "day >= ? AND day <= ?", [week{"from"}, week{"to"}])
    DB.ORDERBY(q, "day")
    DIM rows = DB.ROWS(h, q)
    DB.CLOSE(h)
    RETURN rows
ENDFUNC

' The budget per cost center for one week, from the budgets table.
EXPORT FUNC BUDGETS(db_path$)
    DIM h = OpenDb(db_path$)
    DIM rows = DB.ROWS(h, DB.FROM("budgets"))
    DB.CLOSE(h)
    DIM out AS MAP
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM r = rows[k]
        out{r{"cost_center"}} = r{"week_budget"}
    NEXT k
    RETURN out
ENDFUNC

' Spent, budget and the share used per cost center, largest first: a
' frame with cost_center, spent, budget and percent. A cost center
' with a budget and no bookings shows 0.
EXPORT FUNC REPORT(bookings, budgets)
    DIM spent AS MAP
    IF LEN(bookings) > 0 THEN
        DIM f = DF.FROMROWS(bookings)
        DIM how = {"spent": ["sum", "amount"]}
        DIM sums = DF.GROUPBY(f, "cost_center", how)
        DIM names = DF.COL(sums, "cost_center")
        DIM totals = DF.COL(sums, "spent")
        DIM k = 0
        FOR k = 0 TO LEN(names) - 1
            spent{names[k]} = totals[k]
        NEXT k
    ENDIF
    DIM all_names = UNIQUE(APPEND(MAP.KEYS(spent), MAP.KEYS(budgets)))

```



```

DIM rows = []
DIM i = 0
FOR i = 0 TO LEN(all_names) - 1
    DIM cc$ = all_names[i]
    DIM s = 0.0
    DIM b = 0.0
    IF MAP.EXISTS(spent, cc$) THEN s = spent{cc$}
    IF MAP.EXISTS(budgets, cc$) THEN b = budgets{cc$}
    DIM pct = 0.0
    IF b > 0 THEN pct = ROUND(s / b * 1000) / 10
    PUSH(rows, {"cost_center": cc$, "spent": s, "budget": b, _
        "percent": pct})
NEXT i
RETURN DF.SORT(DF.FROMROWS(rows), "spent", TRUE)
ENDFUNC

' An amount as text with two decimals and a thousands separator.
EXPORT FUNC MONEY$(v)
    DIM whole$ = STR$(INT(ABS(v)))
    DIM cents = ROUND((ABS(v) - INT(ABS(v))) * 100)
    IF cents = 100 THEN
        whole$ = STR$(INT(ABS(v)) + 1)
        cents = 0
    ENDIF
    DIM out$ = ""
    DO WHILE LEN(whole$) > 3
        out$ = "," + RIGHT$(whole$, 3) + out$
        whole$ = LEFT$(whole$, LEN(whole$) - 3)
    LOOP
    out$ = whole$ + out$ + "." + RIGHT$("0" + STR$(cents), 2)
    IF v < 0 THEN out$ = "-" + out$
    RETURN out$
ENDFUNC

' The report as table rows of text, the header first and a total last.
EXPORT FUNC TABLE_ROWS(report)
    DIM rows = [{"Cost center", "Spent", "Budget", "Used"}]
    DIM names = DF.COL(report, "cost_center")
    DIM spent = DF.COL(report, "spent")
    DIM budget = DF.COL(report, "budget")
    DIM pct = DF.COL(report, "percent")
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM used$ = ""
        IF budget[k] > 0 THEN used$ = STR$(pct[k]) + " %"
        DIM s$ = MONEY$(spent[k])
        PUSH(rows, [names[k], s$, MONEY$(budget[k]), used$])
    NEXT k

```

```

    PUSH(rows, ["Total", MONEY$(SUM(spent)), MONEY$(SUM(budget)), ""])
    RETURN rows
ENDFUNC

' The cost centers over budget, as text lines.
EXPORT FUNC OVER_BUDGET(report)
    DIM names = DF.COL(report, "cost_center")
    DIM pct = DF.COL(report, "percent")
    DIM over = FILTER(LAMBDA USE(pct) k -> pct[k] > 100, _
        IOTA(LEN(names)) - 1)
    RETURN SELECT(LAMBDA USE(names, pct) k -> _
        OverLine$(names[k], pct[k]), over)
ENDFUNC

' The line for a cost center over its budget.
FUNC OverLine$(cc$, pct)
    RETURN cc$ + " at " + STR$(pct) + " % of budget"
ENDFUNC

' Spent and budget per cost center as a bar chart, SVG markup.
EXPORT FUNC CHART$(report, week)
    DIM c = SVG.CHART("bar", 640, 360)
    SVG.SETOPT(c, "title", week{"label"})
    SVG.SETOPT(c, "y_title", "EUR")
    SVG.LABELS(c, DF.COL(report, "cost_center"))
    SVG.SERIES(c, "spent", DF.COL(report, "spent"))
    SVG.SERIES(c, "budget", DF.COL(report, "budget"))
    RETURN SVG.MARKUP$(SVG.RENDER(c))
ENDFUNC

FUNC Span$(week)
    RETURN week{"label"} + ", " + week{"from"} + " to " + week{"to"}
ENDFUNC

' The report as a Word file: a heading, the over budget lines and the
' table, for anyone who wants to add comments.
EXPORT SUB WRITE_DOCX(report, week, path$)
    DIM doc = DOCX.NEW()
    DOCX.TITLE(doc, "Team costs")
    DOCX.PARAGRAPH(doc, Span$(week))
    DIM over = OVER_BUDGET(report)
    IF LEN(over) > 0 THEN
        DOCX.HEADING(doc, "Over budget", 2)
        DOCX.BULLETS(doc, over)
    ENDIF
    DOCX.HEADING(doc, "Per cost center", 2)
    DOCX.TABLE(doc, TABLE_ROWS(report), TRUE)
    DIM done = DOCX.WRITE(doc, path$)

```

ENDSUB

' The report as a PDF page: the table and a bar per cost center that
' shows the share of its budget, red when it is over.

```
EXPORT SUB WRITE_PDF(report, week, path$)
    DIM pdf = PDFGEN.DOC()
    PDFGEN.SETINFO(pdf, "Team costs " + week{"from"}, "Report")
    PDFGEN.ADDPAGE(pdf)
    PDFGEN.USEFONT(pdf, "helvetica", "B", 16)
    PDFGEN.TEXTAT(pdf, 20, 25, "Team costs")
    PDFGEN.USEFONT(pdf, "helvetica", "", 10)
    PDFGEN.TEXTAT(pdf, 20, 32, Span$(week))
    PDFGEN.SETXY(pdf, 20, 40)
    DIM rows = TABLE_ROWS(report)
    DIM opts = {"headers": rows[0], "widths": [0, 32, 32, 22], _
        "align": ["L", "R", "R", "R"]}
    PDFGEN.TABLE(pdf, DROP(1, rows), opts)
    DIM y = PDFGEN.POSY(pdf) + 12
    PDFGEN.USEFONT(pdf, "helvetica", "B", 11)
    PDFGEN.TEXTAT(pdf, 20, y, "Share of the budget used")
    PDFGEN.USEFONT(pdf, "helvetica", "", 9)
    DIM names = DF.COL(report, "cost_center")
    DIM pct = DF.COL(report, "percent")
    DIM top = y + 7
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        y = y + 9
        PDFGEN.TEXTAT(pdf, 20, y + 4, names[k])
        DIM w = MIN([pct[k], 150]) * 0.8
        IF pct[k] > 100 THEN
            PDFGEN.SETFILL(pdf, 200, 70, 60)
        ELSE
            PDFGEN.SETFILL(pdf, 78, 121, 167)
        ENDIF
        IF w > 0 THEN PDFGEN.BOX(pdf, 60, y, w, 6, "F")
        PDFGEN.TEXTAT(pdf, 62 + w, y + 4, STR$(pct[k]) + " %")
    NEXT k
    PDFGEN.SETDRAW(pdf, 120, 120, 120)
    PDFGEN.DRAWLINE(pdf, 140, top, 140, y + 8)
    PDFGEN.TEXTAT(pdf, 133, top - 1, "budget")
    PDFGEN.WRITEFILE(pdf, path$)
ENDSUB
```

' The mail that carries the PDF and the Word file.

```
EXPORT FUNC MESSAGE(from$, to$, report, week, pdf$, docx$)
    DIM subject$ = "Team costs, " + week{"label"}
    DIM msg = MAIL.MESSAGE(from$, to$, subject$)
    DIM rows = TABLE_ROWS(report)
```

```

DIM total = rows[LEN(rows) - 1]
DIM LF$ = CHR$(10)
DIM body$ = "Hello," + LF$ + LF$
body$ = body$ + "the team costs of the " + LCASE$(week{"label"})
body$ = body$ + ": " + total[1] + " EUR of "
body$ = body$ + total[2] + " EUR budget." + LF$
DIM over = SELECT(LAMBDA USE(LF$) s$ -> _
    "Over budget: " + s$ + LF$, OVER_BUDGET(report))
body$ = body$ + JOIN(over, "")
body$ = body$ + LF$ + "The report is attached as PDF and Word."
MAIL.PLAIN(msg, body$ + LF$)
MAIL.ATTACH(msg, pdf$)
MAIL.ATTACH(msg, docx$)
RETURN msg
ENDFUNC

' A database with eight weeks of sample bookings on working days up to
' last_day$, and a weekly budget per cost center.
EXPORT SUB SAMPLE(db_path$, last_day$)
    DIM h = DB.OPEN(db_path$)
    DIM booking_cols = {"day": "TEXT", "cost_center": "TEXT", _
        "amount": "REAL"}
    DB.CREATE(h, "bookings", booking_cols)
    DIM budget_cols = {"cost_center": "TEXT", "week_budget": "REAL"}
    DB.CREATE(h, "budgets", budget_cols)
    DIM centers = ["Sales", "Marketing", "IT", "Travel"]
    DIM weekly = [4000, 2500, 3000, 1200]
    DIM plan = []
    DIM c = 0
    FOR c = 0 TO 3
        PUSH(plan, [centers[c], weekly[c]])
    NEXT c
    DIM plan_cols = ["cost_center", "week_budget"]
    DIM b = DB.INSERTMANY(h, "budgets", plan_cols, plan)
    DIM data = []
    DIM d = 0
    FOR d = 0 TO 55
        DIM when = DATEADD("D", -d, CDATE(last_day$ + " 12:00:00"))
        DIM wd = VAL(FORMAT_DATE(when, "%w"))
        DIM day$ = FORMAT_DATE(when, "%Y-%m-%d")
        FOR c = 0 TO 3
            DIM wobble = 0.5 + ((d * 7 + c * 3) MOD 9) / 20
            DIM amount = ROUND(weekly[c] / 5 * wobble)
            IF wd >= 1 AND ALSO wd <= 5 THEN
                PUSH(data, [day$, centers[c], amount])
            ENDIF
        NEXT c
    NEXT d
NEXT d

```

```

DIM cols = ["day", "cost_center", "amount"]
DIM n = DB.INSERTMANY(h, "bookings", cols, data)
DB.CLOSE(h)
ENDSUB

```

How it works

1. **The week.** LAST_WEEK goes back to the Monday of the week the day is in, and seven days more. A run on Monday 5 October reports 28 September to 4 October, and so does a run on the Thursday after, when you start it by hand.
2. **Counting days from noon.** DATEADD counts a day as 24 hours, but the day the clocks change has 23 or 25. Seven days back from midnight can then land at eleven in the evening of the day before, and the report would start on a Sunday. From noon, an hour more or less never changes the date. The test runner of X12 found this mistake in an earlier version of this recipe; that recipe shows the check that caught it.
3. **The database.** BOOKINGS opens the database through RETRY.RUN. When the file is not there yet, because the nightly export writes it last, the first attempt fails, and the pipeline waits a quarter of a second, then half a second, then a second, before it gives up. The query is built with DB.WHERE and question marks; the dates are filled in as values, so no text from outside ever becomes part of the SQL.
4. **The data frame.** REPORT puts the rows into a frame with DF.FROMROWS and adds them up per cost center with DF.GROUPBY. A cost center with a budget and no bookings in the week still gets a row with 0, so the report always has the same lines.
5. **Text for people.** MONEY\$ writes amounts with two decimals and a comma every three digits. TABLE_ROWS turns the frame into rows of text with a total at the end, and every output uses those same rows, so the Word file, the PDF and the mail never disagree.
6. **The chart.** CHART\$ draws spent and budget side by side with SVG.CHART. The PDF draws its own bars with PDFGEN.BOX, because a PDF page takes pictures, not SVG markup.
7. **The log.** LOGGER.TO_FILE writes one line per step into report_pipeline.log in the reports folder, and keeps three old files when it grows past a megabyte. When the database cannot be read, the error goes

into the log before the program stops, so the log says why there was no report.

Run it

```
jdbasic report_pipeline.jdb --dry-run
Week of 28 September 2026: 20 bookings
  Sales: 2,920.00 of 4,000.00
  IT: 2,010.00 of 3,000.00
  Marketing: 1,750.00 of 2,500.00
  Travel: 876.00 of 1,200.00
  Total: 7,556.00 of 10,700.00
would write C:\Users\lena\Documents\AutomateWork\reports\
           costs 2026-09-28.svg, .docx and .pdf
would put a mail to Paul Berger <paul.berger@example.com>

jdbasic report_pipeline.jdb
Wrote C:\Users\lena\Documents\AutomateWork\reports\
      costs 2026-09-28.svg, .docx and .pdf
In the outbox: C:\Users\lena\Documents\AutomateWork\outbox\
               costs 2026-09-28.eml
```

The log of that morning, with its long lines wrapped:

```
2026-10-05 07:00:02 INFO  [report_pipeline] report started
                        from=2026-09-28
2026-10-05 07:00:03 INFO  [report_pipeline] report written
                        base="C:/Users/lena/Documents/AutomateWork/reports\costs
                        2026-09-28" over=0
2026-10-05 07:00:03 INFO  [report_pipeline] mail in the outbox
                        path="C:/Users/lena/Documents/AutomateWork/outbox\costs
                        2026-09-28.eml"
```

--day 2026-10-12 writes the report as if it were that day, which is how you write a week again after finance corrected a booking.

Schedule it

The wizard plans the recipe for Monday at 07:00, every week. If the X01 job server runs your jobs, give the pipeline a job there instead. The server runs its jobs one after another, so the report never starts while a nightly import of the same server still writes, and with attempts it tries again ten minutes later when the database is missing:

```
[[job]]
name = "Monday report"
when = "weekly mon 07:00"
program = "X09_report_pipeline/report_pipeline.jdb"
attempts = 3
retry_seconds = 600
```

The mail waits in the outbox. Look at the PDF, then send it with `jdbasic report_pipeline.jdb --send`. Once the reports have been right for a month, you can let the job server send them too.

Make it yours

- **More than one week.** For a month instead of a week, give `LAST_WEEK` a sister that answers the first and the last day of the month before, and use it in the program instead. Everything after it works on any range.
- **More readers.** `to` takes several addresses, separated by commas:

```
to = "Paul Berger <paul.berger@example.com>, team@example.com"
```

- **Filling the database from a CSV export.** A few lines read the export and append it, here for a file with the columns `day`, `cost_center` and `amount`:

```
DIM rows = CSVREADER("export.csv", ";", TRUE)
DIM h = DB.OPEN(db$)
DIM n = DB.INSERTMANY(h, "bookings", _
    ["day", "cost_center", "amount"], rows)
DB.CLOSE(h)
```

- **A warning line of your own.** `OVER_BUDGET` lists cost centers above 100 percent. To hear about them a little earlier, compare with 90 instead and change the words in `OverLine$` to *close to its budget*.

When it goes wrong

- **"No database at":** the path in database is wrong, or the export has not created the file yet. Run with `--sample` once to see where the pipeline looks.
- **"database is locked":** the pipeline ran while the export was still writing. Start it later, or let the X01 job server run both, one after the other.

- **All amounts are 0:** the dates in the database have another form, such as 05.10.2026. The query compares text, so day must be written YYYY-MM-DD.
- **The report starts on a Sunday:** you changed LAST_WEEK and lost the noon. Run the tests of X12; they say so within a second.

Balance dividend

About 60 minutes a week: 45 on Monday morning, and the quarter of an hour it takes to find out why last week's numbers looked wrong.

X10 Let an AI Drive

The chore

By now Lena's team runs twenty automations. Most days they work. Some days one of them does not, and the question "what happened last night?" means opening the logs folder, finding the right file and reading through it. Lena already uses an AI agent, Claude Code, for her spreadsheets and scripts. She would like to ask it: "Did the backup run last night? What is waiting in the outbox? Try the invoice run for next month and tell me what it would do." The agent can only answer if it can reach the automations, and Lena wants to decide exactly how far it may reach.

What you get

Four tools an AI agent can call, served by the MCP server that is built into `jdBasic`:

Tool	What the agent gets
<code>list_recipes</code>	Your installed automations: id, name, summary, and whether it may run them for real
<code>recipe_log</code>	The last twenty lines of one automation's log
<code>outbox</code>	The mails waiting in the outbox, with recipients and subjects
<code>run_recipe</code>	The output of one automation, run as a dry run

A real run is possible only for the recipes you name in `work.conf`. Nothing is sent from the outbox; the tool only lists it. Every call the agent makes is a line in `logs/ai_drive.jsonl`, with the tool, the arguments and what happened, so you can read afterwards what it did.

MCP, the Model Context Protocol, is the way AI agents such as Claude Code and Claude Desktop find and call tools. An MCP server is a program the agent starts; it lists its tools and answers their calls. `jdBasic` is such a server when you start it with `--mcp`. With `--tools-only` it offers tools of your own and nothing else, each described by a small JSON file that names a `jdBasic` function.

Before you start

You need a `jdBasic` built with the MCP server, and an agent that speaks MCP. The forms release this book starts with has the server; `jdbasic --version` lists *MCP* among its features. The steps below use Claude Code. The setup wizard must have run, because the tools read your recipes and their settings from the work folder.

Watch out

Started with `--tools` instead of `--tools-only`, the `jdBasic` server also offers its own tools, such as `jdb_eval`, which runs any `jdBasic` code the agent writes. With that tool an agent can do whatever you can do on your computer, past every rule of this recipe. The command this recipe prints uses `--tools-only`; keep it that way.

The program

The program writes the four tool files and prints the command that registers the server:

```
' Let an AI Drive: hands your automations to an AI agent such as
' Claude Code through the jdBasic MCP server. It writes one small file
' per tool into the ai-tools folder of your work folder and prints
' the command that tells the agent about the server. The settings are
' the [ai_drive] part of work.conf.
'
'   jdbasic ai_drive.jdb           write the tool files
'   jdbasic ai_drive.jdb --dry-run show them, write nothing
'   jdbasic ai_drive.jdb --check  ask the server for its tools

IMPORT WORKCONF, AITOOLS

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM home$ = "~/Documents/AutomateWork"
DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
DIM jdbasic$ = WORKCONF.VALUE(cfg, "paths.jdbasic", "jdbasic")
DIM tools$ = PATH.JOIN$(work$, "ai-tools")
DIM own$ = PATH.JOIN$(work$, "recipes", "X10_ai_drive")
DIM mine$ = PATH.JOIN$(own$, "aitools.jdb")
DIM module$ = WORKCONF.FOLDER$(cfg, "ai_drive.module", mine$)
DIM q$ = CHR$(34)

IF WORKCONF.FLAG("check") THEN
```

```

    DIM offered = AITTOOLS.CHECK(jdbasic$, tools$, work$)
    IF LEN(offered) = 0 THEN
        PRINT "The server did not answer. Is "; jdbasic$; " right?"
        END 1
    ENDIF
    PRINT "The server offers: "; JOIN(offered, ", ")
    END
ENDIF

IF NOT FILE.EXISTS(module$) THEN
    PRINT "No tool module at "; module$
    END 1
ENDIF
DIM dry_run = WORKCONF.FLAG("dry-run")
IF NOT dry_run THEN MKDIR tools$
DIM manifests = AITTOOLS.MANIFESTS(module$)
DIM k = 0
FOR k = 0 TO LEN(manifests) - 1
    DIM m = manifests[k]
    DIM file$ = PATH.JOIN$(tools$, m{"name"} + ".json")
    IF dry_run THEN
        PRINT "would write "; file$
    ELSE
        TXTWRITER file$, JSON.STRINGIFY$(m)
        PRINT "wrote "; file$
    ENDIF
NEXT k

DIM allowed = WORKCONF.VALUE(cfg, "ai_drive.allow", [])
PRINT "Real runs allowed for: "; LEN(allowed); " recipe(s)"
PRINT "Tell Claude Code about the server once, in a command prompt:"
DIM line1$ = "claude mcp add automate-work -e AUTOMATEWORK_CONF="
line1$ = line1$ + q$ + WORKCONF.CONFIG_PATH$() + q$
DIM line2$ = "-- " + q$ + jdbasic$ + q$ + " --mcp --tools-only "
line2$ = line2$ + q$ + tools$ + q$
PRINT " "; line1$
PRINT " "; line2$

```

The module holds the tools themselves, the tool files and a check that asks the server which tools it offers:

```

' aitools.jdb: the tools an AI agent may use on your automations,
' served by the jdBasic MCP server. Each TOOL_ function takes the
' arguments as JSON text and answers text. An agent can list the
' recipes, read their logs, look into the outbox and run a recipe,
' as a dry run unless work.conf allows that recipe to run for real.
' Every call is written to logs/ai_drive.jsonl.

```

```

'
'   jdbasic --mcp --tools-only <work folder>/ai-tools

EXPORT MODULE AITOOLS

IMPORT WORKCONF, CONF, OUTBOX, MAIL, LOGGER

DIM gLF$ = CHR$(10)

' The settings of work.conf, read again for every call.
FUNC Settings()
    RETURN WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
ENDFUNC

FUNC WorkDir$(cfg)
    DIM home$ = "~/Documents/AutomateWork"
    RETURN WORKCONF.FOLDER$(cfg, "paths.work", home$)
ENDFUNC

' The installed recipes as maps with id, name, summary, folder,
' program and section, read from their recipe.toml files.
EXPORT FUNC RECIPES(work$)
    DIM root$ = PATH.JOIN$(work$, "recipes")
    DIM out = []
    IF NOT FILE.ISDIR(root$) THEN RETURN out
    DIM names = SORT(DIR$(PATH.JOIN$(root$, "*")))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM entry$ = names[k]
        DIM folder$ = PATH.JOIN$(root$, entry$)
        DIM toml$ = PATH.JOIN$(folder$, "recipe.toml")
        IF FILE.EXISTS(toml$) THEN
            DIM t = CONF.TOML(toml$)
            DIM r AS MAP
            r{"id"} = CONF.GET(t, "id", "")
            r{"name"} = CONF.GET(t, "name", "")
            r{"summary"} = CONF.GET(t, "summary", "")
            r{"program"} = CONF.GET(t, "program", "")
            r{"section"} = CONF.GET(t, "section", "")
            r{"folder"} = folder$
            PUSH(out, r)
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

FUNC FindRecipe(recipes, id$)
    DIM hits = FILTER(LAMBDA USE(id$) r -> _

```

```

        UCASE$(r{"id"}) = UCASE$(id$), recipes)
    DIM hint$ = "; list_recipes names them."
    IF LEN(hits) = 0 THEN THROW "No recipe " + id$ + hint$
    DIM hit AS MAP
    hit = hits[LEN(hits) - 1]
    RETURN hit
ENDFUNC

' TRUE when [ai_drive] allow names the recipe for real runs.
EXPORT FUNC ALLOWED(cfg, id$)
    DIM allow = CONF.GET(cfg, "ai_drive.allow", [])
    RETURN UCASE$(id$) IN SELECT(LAMBDA s$ -> UCASE$(s$), allow)
ENDFUNC

' Writes one line about a tool call to logs/ai_drive.jsonl.
SUB Audit(cfg, tool$, args$, outcome$)
    DIM logs$ = PATH.JOIN$(WorkDir$(cfg), "logs")
    MKDIR logs$
    DIM lg = LOGGER.NEW("ai_drive")
    LOGGER.TO_JSONL(lg, PATH.JOIN$(logs$, "ai_drive.jsonl"))
    DIM fields AS MAP
    fields{"tool"} = tool$
    fields{"args"} = args$
    LOGGER.INFO(lg, outcome$, fields)
ENDSUB

' list_recipes: every installed recipe with id, name and summary.
EXPORT FUNC TOOL_LIST(args$)
    DIM cfg = Settings()
    DIM all = RECIPES(WorkDir$(cfg))
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(all) - 1
        DIM r = all[k]
        DIM one AS MAP
        one{"id"} = r{"id"}
        one{"name"} = r{"name"}
        one{"summary"} = r{"summary"}
        one{"real_runs"} = ALLOWED(cfg, r{"id"})
        PUSH(out, one)
    NEXT k
    Audit(cfg, "list_recipes", args$, STR$(LEN(out)) + " recipes")
    RETURN JSON.STRINGIFY$(out)
ENDFUNC

' recipe_log: the last lines of a recipe's log file.
EXPORT FUNC TOOL_LOG(args$)
    DIM cfg = Settings()

```

```

    DIM a = JSON.PARSE$(args$)
    DIM r = FindRecipe(RECIPES(WorkDir$(cfg)), a{"id"})
    DIM name$ = r{"section"} + ".log"
    DIM path$ = PATH.JOIN$(WorkDir$(cfg), "logs", name$)
    Audit(cfg, "recipe_log", args$, r{"id"})
    IF NOT FILE.EXISTS(path$) THEN RETURN "No log yet for " + r{"id"}
    DIM rows = SPLIT(TRIM$(TXTREADER$(path$)), gLF$)
    DIM keep = 20
    IF LEN(rows) < keep THEN keep = LEN(rows)
    RETURN JOIN(TAKE(-keep, rows), gLF$)
ENDFUNC

' outbox: the mails that wait in the outbox. Nothing is sent.
EXPORT FUNC TOOL_OUTBOX(args$)
    DIM cfg = Settings()
    DIM files = OUTBOX.PENDING(cfg)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(files) - 1
        DIM path$ = files[k]
        DIM m = MAIL.PARSEFILE(path$)
        DIM one AS MAP
        one{"file"} = PATH.BASENAME$(path$)
        one{"to"} = JOIN(m{"to"}, ", ")
        one{"subject"} = m{"subject"}
        PUSH(out, one)
    NEXT k
    Audit(cfg, "outbox", args$, STR$(LEN(out)) + " mails")
    RETURN JSON.STRINGIFY$(out)
ENDFUNC

' run_recipe: runs a recipe and answers its output. It is a dry run
' unless for_real is true and [ai_drive] allow names the recipe.
EXPORT FUNC TOOL_RUN(args$)
    DIM cfg = Settings()
    DIM a = JSON.PARSE$(args$)
    DIM r = FindRecipe(RECIPES(WorkDir$(cfg)), a{"id"})
    DIM for_real = FALSE
    IF MAP.EXISTS(a, "for_real") THEN for_real = a{"for_real"}
    IF for_real AND ALSO NOT ALLOWED(cfg, r{"id"}) THEN
        Audit(cfg, "run_recipe", args$, "refused")
        RETURN r{"id"} + " may only run as a dry run. Ask the person."
    ENDIF
    DIM jdbasic$ = WORKCONF.VALUE(cfg, "paths.jdbasic", "jdbasic")
    DIM program$ = PATH.JOIN$(r{"folder"}, r{"program"})
    DIM argv = [program$, "--config", WORKCONF.CONFIG_PATH$()]
    IF NOT for_real THEN PUSH(argv, "--dry-run")
    DIM res = OS.EXEC(jdbasic$, argv)

```

```

    DIM code = res{"EXIT_CODE"}
    DIM text$ = LEFT$(res{"OUTPUT"}, 4000)
    DIM how$ = "dry run"
    IF for_real THEN how$ = "real run"
    Audit(cfg, "run_recipe", args$, how$ + ", exit " + STR$(code))
    DIM head$ = how$ + " of " + r{"id"} + ", exit " + STR$(code)
    RETURN head$ + gLF$ + text$
ENDFUNC

' The manifests the MCP server reads, one per tool, for the module
' at module$.
EXPORT FUNC MANIFESTS(module$)
    DIM none AS MAP
    DIM by_id AS MAP
    DIM id_field AS MAP
    id_field{"type"} = "string"
    id_field{"description"} = "Recipe id, like E11"
    by_id{"id"} = id_field
    DIM run_args AS MAP
    run_args{"id"} = by_id{"id"}
    run_args{"for_real"} = {"type": "boolean", _
        "description": "Run for real; only for allowed recipes"}
    DIM out = []
    PUSH(out, Manifest("list_recipes", "TOOL_LIST", module$, _
        "The person's automations: id, name, summary.", none, []))
    PUSH(out, Manifest("recipe_log", "TOOL_LOG", module$, _
        "The last lines of one automation's log.", by_id, ["id"]))
    PUSH(out, Manifest("outbox", "TOOL_OUTBOX", module$, _
        "Mails waiting in the outbox. Nothing is sent.", none, []))
    PUSH(out, Manifest("run_recipe", "TOOL_RUN", module$, _
        "Runs an automation, as a dry run unless allowed.", _
        run_args, ["id"]))
    RETURN out
ENDFUNC

FUNC Manifest(name$, handler$, module$, about$, props, needed)
    DIM schema AS MAP
    schema{"type"} = "object"
    schema{"properties"} = props
    IF LEN(needed) > 0 THEN schema{"required"} = needed
    DIM m AS MAP
    m{"name"} = name$
    m{"description"} = about$
    m{"module"} = REPLACE$(module$, "\", "/" )
    m{"handler"} = handler$
    m{"inputSchema"} = schema
    RETURN m
ENDFUNC

```

```

' The tool names the MCP server offers for tools$, asked over stdio
' the way an agent asks. scratch$ is a folder for the request file.
EXPORT FUNC CHECK(jdbasic$, tools$, scratch$)
    DIM q$ = CHR$(34)
    DIM hello$ = "{" + q$ + "jsonrpc" + q$ + ":" + q$ + "2.0" + q$
    DIM ask$ = "," + q$ + "method" + q$ + ":" + q$
    DIM init$ = hello$ + "," + q$ + "id" + q$ + ":1" + ask$
    init$ = init$ + "initialize" + q$ + "}"
    DIM list$ = hello$ + "," + q$ + "id" + q$ + ":2" + ask$
    list$ = list$ + "tools/list" + q$ + "}"
    DIM req$ = PATH.JOIN$(scratch$, "mcp_check.txt")
    TXTWRITER req$, init$ + gLF$ + list$ + gLF$
    DIM argv = ["/c", jdbasic$, "--mcp", "--tools-only", tools$]
    PUSH(argv, "<")
    PUSH(argv, req$)
    DIM res = OS.EXEC("cmd", argv)
    KILL req$
    DIM rows = SPLIT(res{"OUTPUT"}, gLF$)
    DIM names = []
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM row$ = TRIM$(rows[k])
        ' The server writes one reply per line; what it says
        ' about itself is not JSON and is not a reply.
        IF LEFT$(row$, 1) <> "{" THEN CONTINUEFOR
        DIM reply = JSON.PARSE$(row$)
        IF TYPEOF(reply) = "OBJECT" AND ALSO IsListReply(reply) THEN
            DIM tools = reply{"result"}{"tools"}
            DIM j = 0
            FOR j = 0 TO LEN(tools) - 1
                PUSH(names, tools[j]{"name"})
            NEXT j
        ENDIF
    NEXT k
    RETURN names
ENDFUNC

FUNC IsListReply(reply)
    IF NOT MAP.EXISTS(reply, "id") THEN RETURN FALSE
    IF reply{"id"} <> 2 THEN RETURN FALSE
    RETURN MAP.EXISTS(reply, "result")
ENDFUNC

```


How it works

1. **How the server finds a tool.** For every `.json` file in the folder after `--tools`, the server reads the tool's name, its description, the arguments it takes and the `jdBasic` function that handles it. It runs the module file once at start, and for each call it hands the function the arguments as JSON text and passes the text it answers back to the agent.
2. **The tool files.** `MANIFESTS` builds the four descriptions. `Manifest` puts each one together: name, description, an `inputSchema` that tells the agent which arguments exist, the module and the handler. The descriptions are short on purpose. An agent reads them to decide which tool fits a question, so they say what the tool does in plain words.
3. **Fresh settings for every call.** Each tool reads `work.conf` again, so a change you make while the agent runs counts from the next call on. `RECIPES` reads the `recipe.toml` of every recipe the wizard installed in the work folder.
4. **Reading only.** `TOOL_LIST`, `TOOL_LOG` and `TOOL_OUTBOX` read and answer. `TOOL_OUTBOX` uses `OUTBOX.PENDING`, the same function the recipes of Chapter 4 use, and opens each mail only to read its recipients and subject.
5. **A dry run unless allowed.** `TOOL_RUN` starts the recipe's program with the `jdbasic` the wizard found, with `--config` and with `--dry-run`. Only when the agent asks for a real run and `ALLOWED` finds the recipe's id in `[ai_drive]` allow does it leave out `--dry-run`. Otherwise it refuses and tells the agent to ask you. The answer is the program's output, cut to 4000 characters.
6. **The audit trail.** `Audit` writes one JSON line per call into `logs/ai_drive.jsonl` through the `LOGGER` library: time, tool, arguments and outcome, the refusals included.
7. **The check.** `CHECK` starts the server the way an agent would, sends it the two requests an agent sends first, and reads the names in the answer. That is what `--check` prints.

Run it

Write the tool files and get the command for Claude Code:

```
jdbasic ai_drive.jdb
wrote C:\Users\lena\Documents\AutomateWork\ai-tools\list_recipes.json
wrote C:\Users\lena\Documents\AutomateWork\ai-tools\recipe_log.json
```

```
wrote C:\Users\lena\Documents\AutomateWork\ai-tools\outbox.json
wrote C:\Users\lena\Documents\AutomateWork\ai-tools\run_recipe.json
Real runs allowed for: 0 recipe(s)
Tell Claude Code about the server once, in a command prompt:
  claude mcp add automate-work -e AUTOMATEWORK_CONF="C:\Users\..."
    -- "C:\Users\lena\jdBasic\jdbasic.exe" --mcp --tools-only "C:\..."
```

Type the two printed lines as one command. Then check that the server starts and offers the tools:

```
jdbasic ai_drive.jdb --check
The server offers: list_recipes, outbox, recipe_log, run_recipe
```

Start Claude Code and ask, for example, "Which of my automations wrote an error into its log this week?" The agent calls `list_recipes`, then `recipe_log` for each recipe, and answers from what the logs say. Each call appears in `logs/ai_drive.jsonl`:

```
{"time":"2026-10-02 07:20:28","level":"INFO","logger":"ai_drive",
  "message":"refused","tool":"run_recipe",
  "args":{"id":"X10","for_real":true}}
```

Schedule it

There is nothing to schedule. The agent starts the server when it needs it and stops it when it is done. Run `ai_drive.jdb` again only when you move the work folder, change the module or add a tool.

Make it yours

The setting is in the part of `work.conf` that starts with `[ai_drive]`:

```
[ai_drive]
allow = ["E11", "M08"]
```

- **Real runs.** `allow` names the recipes the agent may run for real. Start with recipes that only read and report, such as the Disk Space Report (E11) or the Personal Dashboard (M08). A recipe that moves files or writes mail is better left to a dry run, with you deciding.

- **Ask before every call.** Claude Code asks before it calls a tool. Allow `list_recipes`, `recipe_log` and `outbox` for good, since they only read, and keep answering each `run_recipe` request yourself.
- **A tool of your own.** Write a function in `aitools.jdb` that takes the arguments as text and answers text, such as `TOOL_HOURS` that reads this week's hours from the Time Tracker's log (E07). Add a Manifest line for it in `MANIFESTS`, run `ai_drive.jdb` again and restart the agent.
- **Another agent.** Claude Desktop and other MCP clients start the same server with the same command; only the place where you enter it differs. In Claude Desktop it goes into its configuration file under `mcpServers`.

When it goes wrong

- **--check lists only the jdb tools:** the server could not load `aitools.jdb`. Run the server by hand with the variable `JDBASIC_MCP_LOG=1` set; it then reports on the error output which module it tried and why it failed. A missing module such as `WORKCONF` means the wizard's library folder `.jdbasic\lib` is missing.
- **"The server did not answer":** `jdbasic` in `[paths]` of `work.conf` does not point to a `jdBasic` with the MCP server. Run `jdbasic --version` and look for `MCP` among the features.
- **"No recipe X":** the agent used an id that is not installed. It usually asks `list_recipes` next and corrects itself.
- **A dry run that changes something:** a recipe without a dry run of its own ignores `--dry-run`. All recipes of this book have one; for a recipe of your own, check it before you let an agent run it.

Balance dividend

About 20 minutes a week of reading logs and checking runs, given to an agent that does it when you ask, inside the limits you set.

X11 Web Harvester

The chore

Every Monday Lena updates a sheet with the prices of the supplies her department buys most: copy paper, toner, the coffee for the kitchen. The purchasing team wants to know when a supplier raises a price, and the numbers are on the suppliers' web shops, one page per product. So she opens fourteen pages, finds the price on each, types it into Excel and checks that the comma ended up in the right place.

What you get

A CSV file with one line per value and run: the time, the name you gave the value, the value as the page shows it, the same value as a plain number, the address, and *ok* or what went wrong. Excel opens the file, and a chart over the weeks shows when a price moved.

The harvester reads the pages the way a considerate visitor does. It asks each site's `robots.txt` which pages it may read, waits between two requests to the same site, keeps a page it has read for twelve hours instead of asking again, and gives a busy server a second and a third chance before it reports the page as failed.

Before you start

Pick the pages and find the element that holds each value:

1. Open the page in your browser, right-click the price and choose *Inspect*. The browser shows the HTML around it, for example `<span class="price">4,99 €</span>`.
2. Write a CSS selector for it: `span.price` means a `span` element with the class `price`. If the page has several prices, make it more precise, such as `div.product-main span.price`.
3. Put each value into `targets.toml` in the harvester's folder, as in the example file that comes with the recipe:

```
[[target]]
name = "Copy paper A4, Office Shop"
url = "https://shop.example.com/paper/a4-80g"
select = "div.product-main span.price"

[[target]]
name = "Toner X200 data sheet"
url = "https://shop.example.com/toner/x200"
select = "a.datasheet"
attr = "href"
```

`attr` reads an attribute of the element instead of its text, here the address of a data sheet.

Watch out

Being polite to a server is not the same as being allowed to read it. Many shops say in their terms of use whether automated reading is permitted, and some offer a price list or an interface for it. Prefer those, read the terms before you harvest a site, and never collect personal data this way.

The program

The program reads the targets, runs the harvester and adds the lines to the CSV file:

```
' Web Harvester: values such as prices read from web pages into a CSV
' file that opens in Excel, politely: robots.txt is asked first, each
' host gets a pause between two requests, and a page read in the last
' hours comes from the cache. targets.toml names the pages and what
' to read from them; the settings are the [web_harvester] part of
' work.conf.
'
' jdbasic web_harvester.jdb          read all targets
' jdbasic web_harvester.jdb --dry-run read and show, write nothing

IMPORT WORKCONF, HARVEST, CONF, LOGGER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "web_harvester."
DIM home$ = "~/Documents/AutomateWork/harvest"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", home$)
DIM own$ = PATH.JOIN$(PATH.DIRNAME$(OS.ARGS()[0]), "targets.toml")
DIM targets$ = WORKCONF.FOLDER$(cfg, key$ + "targets", _
```

```

    PATH.JOIN$(folder$, "targets.toml"))
IF NOT FILE.EXISTS(targets$) THEN targets$ = own$
DIM spec = CONF.TOML(targets$)
IF NOT MAP.EXISTS(spec, "target") THEN
    THROW "No [[target]] in " + targets$
ENDIF
DIM targets = spec{"target"}
DIM delay = WORKCONF.VALUE(cfg, key$ + "delay_seconds", 5)
DIM hours = WORKCONF.VALUE(cfg, key$ + "cache_hours", 12)
DIM agent$ = "AutomateWork harvester (jdBasic)"
DIM contact$ = WORKCONF.VALUE(cfg, "user.email", "")
IF contact$ <> "" THEN agent$ = agent$ + "; " + contact$
DIM output$ = PATH.JOIN$(folder$, "harvest.csv")
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM lg = LOGGER.NEW("web_harvester")
LOGGER.TO_CONSOLE(lg, FALSE)
IF NOT dry_run THEN
    MKDIR folder$
    LOGGER.TO_FILE(lg, PATH.JOIN$(folder$, "harvest.log"), 1000000, 2)
ENDIF

DIM opts = {"agent": agent$, "delay": delay, "hours": hours}
opts{"cache"} = PATH.JOIN$(folder$, "cache.json")
DIM h = HARVEST.NEW(opts)
DIM now$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M:%S")
DIM rows = HARVEST.RUN(h, targets, now$)

DIM failed = 0
DIM lines = []
DIM k = 0
FOR k = 0 TO LEN(rows) - 1
    DIM row = rows[k]
    IF row[5] <> "ok" THEN
        failed = failed + 1
        LOGGER.WARN(lg, row[1] + ": " + row[5])
    ELSEIF dry_run THEN
        PRINT row[1]; ": "; row[2]; " ("; row[3]; ")"
    ENDIF
    PUSH(lines, HARVEST.CSVLINE$(row))
NEXT k
IF dry_run THEN
    PRINT LEN(rows) - failed; " values read; nothing written"
END
ENDIF

DIM bom$ = CHR$(239) + CHR$(187) + CHR$(191)
DIM text$ = bom$ + "time,name,value,number,url,status" + CHR$(10)

```

```

IF FILE.EXISTS(output$) THEN text$ = TXTREADER$(output$)
TXTWRITER output$, text$ + JOIN(lines, CHR$(10)) + CHR$(10)
HARVEST.SAVE(h, opts{"cache"})
LOGGER.INFO(lg, "harvest written", {"values": LEN(rows) - failed, _
    "failed": failed, "requests": h{"requests"}})

```

The module holds the politeness and the reading:

```

' harvest.jdb: the work of the Web Harvester. It reads values such as
' prices from web pages the way a polite visitor does: it asks each
' site's robots.txt first, waits between two requests to the same
' host, keeps pages it has read for a while, and tries a busy server
' again a little later.
'
'   IMPORT HARVEST
'   DIM h = HARVEST.NEW({"agent": "AutomateWork", "delay": 5})
'   DIM rows = HARVEST.RUN(h, targets, "2026-10-05 07:00:00")
'   HARVEST.SAVE(h, "cache.json")

EXPORT MODULE HARVEST

IMPORT REQ, CACHE, RETRY, HTMLDOM

DIM gH AS MAP

' — robots.txt —————

' The rules of a robots.txt for agent$: the group that names the
' agent, else the group for every agent. A map of "allow" and
' "disallow" (lists of path patterns) and "delay" (Crawl-delay in
' seconds, 0 when none).
EXPORT FUNC ROBOTS(text$, agent$)
    DIM me$ = LCASE$(agent$)
    DIM mine = {"allow": [], "disallow": [], "delay": 0}
    mine{"found"} = FALSE
    DIM star = {"allow": [], "disallow": [], "delay": 0}
    star{"found"} = FALSE
    DIM for_me = FALSE
    DIM for_all = FALSE
    DIM in_rules = FALSE
    DIM lines = SPLIT(REPLACE$(text$, CHR$(13), ""), CHR$(10))
    DIM k = 0
    FOR k = 0 TO LEN(lines) - 1
        DIM line$ = lines[k]
        DIM hash = INSTR(line$, "#")
        IF hash >= 0 THEN line$ = LEFT$(line$, hash)
        DIM colon = INSTR(line$, ":")

```

```

    IF colon > 0 THEN
        DIM field$ = LCASE$(TRIM$(LEFT$(line$, colon)))
        DIM value$ = TRIM$(MID$(line$, colon + 1, LEN(line$)))
        IF field$ = "user-agent" THEN
            IF in_rules THEN
                for_me = FALSE
                for_all = FALSE
                in_rules = FALSE
            ENDIF
            DIM who$ = LCASE$(value$)
            IF who$ = "*" THEN
                for_all = TRUE
            ELSEIF who$ <> "" ANDALSO INSTR(me$, who$) >= 0 THEN
                for_me = TRUE
            ENDIF
        ELSE
            in_rules = TRUE
            IF for_me THEN AddRule(mine, field$, value$)
            IF for_all THEN AddRule(star, field$, value$)
        ENDIF
    ENDIF
NEXT k
IF mine{"found"} THEN RETURN mine
RETURN star
ENDFUNC

SUB AddRule(group, field$, value$)
    group{"found"} = TRUE
    IF field$ = "crawl-delay" THEN group{"delay"} = VAL(value$)
    IF value$ = "" THEN RETURN
    IF field$ = "allow" ORELSE field$ = "disallow" THEN
        DIM list = group{field$}
        PUSH(list, value$)
    ENDIF
ENDSUB

' Whether a robots.txt path pattern matches path$: a prefix, where *
' stands for any characters and a final $ for the end of the path.
EXPORT FUNC MATCHES(pattern$, path$)
    DIM anchored = ENDSWITH(pattern$, "$")
    DIM pat$ = pattern$
    IF anchored THEN pat$ = LEFT$(pat$, LEN(pat$) - 1)
    DIM parts = SPLIT(pat$, "*")
    DIM first$ = parts[0]
    IF NOT STARTSWITH(path$, first$) THEN RETURN FALSE
    DIM at = LEN(first$)
    DIM k = 0
    FOR k = 1 TO LEN(parts) - 1

```



```

        DIM part$ = parts[k]
        DIM rest$ = MID$(path$, at, LEN(path$))
        DIM found = INSTR(rest$, part$)
        IF found < 0 THEN RETURN FALSE
        at = at + found + LEN(part$)
    NEXT k
    IF NOT anchored THEN RETURN TRUE
    IF LEN(parts) = 1 THEN RETURN path$ = first$
    RETURN ENDSWITH(path$, parts[LEN(parts) - 1])
ENDFUNC

' Whether the rules let the agent read path$: the longest matching
' pattern decides, and Allow wins a tie.
EXPORT FUNC ALLOWED(rules, path$)
    DIM best = -1
    DIM verdict = TRUE
    DIM kinds = ["disallow", "allow"]
    DIM j = 0
    DIM k = 0
    FOR j = 0 TO 1
        DIM list = rules{kinds[j]}
        FOR k = 0 TO LEN(list) - 1
            DIM pat$ = list[k]
            IF MATCHES(pat$, path$) AND ALSO LEN(pat$) >= best THEN
                best = LEN(pat$)
                verdict = FALSE
                IF j = 1 THEN verdict = TRUE
            ENDIF
        NEXT k
    NEXT j
    RETURN verdict
ENDFUNC

' — fetching —————

FUNC SiteOf$(url$)
    DIM after = INSTR(url$, "://") + 3
    DIM slash = INSTR(MID$(url$, after, LEN(url$)), "/")
    IF slash < 0 THEN RETURN url$ + "/"
    RETURN LEFT$(url$, after + slash + 1)
ENDFUNC

FUNC PathOf$(url$)
    DIM site$ = SiteOf$(url$)
    IF LEN(url$) < LEN(site$) THEN RETURN "/"
    RETURN MID$(url$, LEN(site$) - 1, LEN(url$))
ENDFUNC

```

```

' Waits until the last request to the host is the pause ago.
SUB Pause(site$)
    DIM gap = gH{"delay"} * 1000
    DIM robots = gH{"robots"}
    IF MAP.EXISTS(robots, site$) THEN
        DIM rules = robots{site$}
        DIM asked = rules{"delay"} * 1000
        IF asked > gap THEN gap = asked
    ENDIF
    DIM last = gH{"last"}
    IF MAP.EXISTS(last, site$) THEN
        DIM since = TICK() - last{site$}
        IF since < gap THEN SLEEP gap - since
    ENDIF
ENDSUB

FUNC Transient(message$)
    RETURN STARTSWITH(message$, "transient")
ENDFUNC

' One request; a busy or unreachable server raises a "transient"
' error that RETRY tries again.
FUNC GetOnce(url$)
    DIM site$ = SiteOf$(url$)
    Pause(site$)
    DIM r = REQ.GET(gH{"sess"}, url$)
    DIM last = gH{"last"}
    last{site$} = TICK()
    gH{"requests"} = gH{"requests"} + 1
    DIM status = r{"status"}
    IF status = 0 ORELSE status = 429 ORELSE status >= 500 THEN
        THROW "transient: the server answered " + STR$(status)
    ENDIF
    DIM out AS MAP
    out{"status"} = status
    out{"body"} = r{"body"}
    RETURN out
ENDFUNC

FUNC Get(url$)
    DIM opts = {"attempts": 3, "delay": gH{"retry_ms"}}
    opts{"retry_if"} = Transient@
    DIM report = RETRY.ATTEMPT(GetOnce@, url$, opts)
    IF NOT report{"ok"} THEN
        DIM why$ = report{"error"}
        THROW why$
    ENDIF
    RETURN report{"value"}

```

```
ENDFUNC
```

```
' The robots rules of the site of url$, read once per run. A site
' without a robots.txt allows everything.
```

```
FUNC RulesFor(url$)
    DIM site$ = SiteOf$(url$)
    DIM robots = gH{"robots"}
    IF MAP.EXISTS(robots, site$) THEN RETURN robots{site$}
    DIM r = Get(site$ + "robots.txt")
    DIM text$ = ""
    IF r{"status"} = 200 THEN text$ = r{"body"}
    DIM rules = ROBOTS(text$, gH{"agent"})
    robots{site$} = rules
    RETURN rules
```

```
ENDFUNC
```

```
' A harvester. opts: "agent" (the name it sends and looks for in
' robots.txt), "delay" (seconds between two requests to a host),
' "cache" (a file of pages read before), "hours" (how long a page is
' kept) and "retry_ms" (the first wait before another try).
```

```
EXPORT FUNC NEW(opts)
    DIM h AS MAP
    h{"agent"} = opts{"agent"}
    h{"delay"} = opts{"delay"}
    h{"retry_ms"} = 2000
    IF MAP.EXISTS(opts, "retry_ms") THEN
        h{"retry_ms"} = opts{"retry_ms"}
    ENDIF
    DIM hours = 12
    IF MAP.EXISTS(opts, "hours") THEN hours = opts{"hours"}
    DIM keep = {"ttl": hours * 3600}
    DIM path$ = ""
    IF MAP.EXISTS(opts, "cache") THEN path$ = opts{"cache"}
    IF path$ <> "" AND ALSO FILE.EXISTS(path$) THEN
        h{"cache"} = CACHE.LOAD(path$, keep)
    ELSE
        h{"cache"} = CACHE.NEW(keep)
    ENDIF
    DIM sess = REQ.NEW("")
    REQ.HEADER(sess, "User-Agent", h{"agent"})
    REQ.TIMEOUT(sess, 30)
    h{"sess"} = sess
    h{"robots"} = {}
    h{"last"} = {}
    h{"requests"} = 0
    RETURN h
```

```
ENDFUNC
```

```

' The page at url$: from the cache while it is fresh, else read from
' the site if its robots.txt allows it.
EXPORT FUNC PAGE$(h, url$)
    gH = h
    DIM cache = h{"cache"}
    IF CACHE.HAS(cache, url$) THEN RETURN CACHE.FETCH(cache, url$, "")
    DIM rules = RulesFor(url$)
    IF NOT ALLOWED(rules, PathOf$(url$)) THEN
        THROW "robots.txt asks to stay out of " + PathOf$(url$)
    ENDIF
    DIM r = Get(url$)
    IF r{"status"} <> 200 THEN
        THROW "the page answered " + STR$(r{"status"})
    ENDIF
    DIM body$ = r{"body"}
    CACHE.PUT(cache, url$, body$)
    RETURN body$
ENDFUNC

' Writes the pages read to the cache file.
EXPORT SUB SAVE(h, path$)
    MKDIR PATH.DIRNAME$(path$)
    CACHE.SAVE(h{"cache"}, path$)
ENDSUB

' — reading values —————

' The text of the first element selector$ finds in html$, or its
' attribute attr$ when that is not "".
EXPORT FUNC VALUE$(html$, selector$, attr$)
    DIM doc = HTMLDOM.PARSE(html$)
    DIM at = HTMLDOM.FIND(doc, selector$)
    IF at < 0 THEN THROW "nothing on the page matches " + selector$
    IF attr$ <> "" THEN RETURN TRIM$(HTMLDOM.ATTR$(doc, at, attr$))
    RETURN TRIM$(REGEX.REPLACE("\s+", HTMLDOM.FLAT$(doc, at), " "))
ENDFUNC

' The first number in a text as plain digits with a point, so
' "1.234,50 EUR" and "$1,234.50" both give "1234.50"; "" when there
' is none.
EXPORT FUNC NUMBER$(text$)
    DIM digits$ = "0123456789"
    DIM start = -1
    DIM k = 0
    FOR k = 0 TO LEN(text$) - 1
        DIM c$ = MID$(text$, k, 1)
        IF start < 0 AND ALSO INSTR(digits$, c$) >= 0 THEN start = k
    NEXT k

```

```

IF start < 0 THEN RETURN ""
DIM n$ = ""
k = start
DIM keep$ = digits$ + ".,'"
DIM ch$ = MID$(text$, k, 1)
DO WHILE k < LEN(text$) ANDALSO INSTR(keep$, ch$) >= 0
    n$ = n$ + ch$
    k = k + 1
    ch$ = MID$(text$, k, 1)
LOOP
n$ = REPLACE$(n$, "'", "")
DO WHILE ENDSWITH(n$, ".") ORELSE ENDSWITH(n$, ",")
    n$ = LEFT$(n$, LEN(n$) - 1)
LOOP
DIM last_dot = -1
DIM last_comma = -1
FOR k = 0 TO LEN(n$) - 1
    IF MID$(n$, k, 1) = "." THEN last_dot = k
    IF MID$(n$, k, 1) = "," THEN last_comma = k
NEXT k
DIM sep = last_dot
IF last_comma > sep THEN sep = last_comma
IF sep < 0 THEN RETURN n$
DIM tail$ = MID$(n$, sep + 1, LEN(n$))
DIM both = last_dot >= 0 ANDALSO last_comma >= 0
IF NOT both ANDALSO LEN(tail$) = 3 THEN
    RETURN REPLACE$(REPLACE$(n$, ".", ""), ",", "")
ENDIF
DIM head$ = REPLACE$(REPLACE$(LEFT$(n$, sep), ".", ""), ",", "")
RETURN head$ + "." + tail$
ENDFUNC

' One row per target, a map with name, url, select and optionally
' attr: time, name, value, number, url and "ok" or what went wrong.
' A target that fails does not stop the others.
EXPORT FUNC RUN(h, targets, now$)
    DIM rows = []
    DIM k = 0
    FOR k = 0 TO LEN(targets) - 1
        DIM t = targets[k]
        DIM url$ = t{"url"}
        DIM attr$ = ""
        IF MAP.EXISTS(t, "attr") THEN attr$ = t{"attr"}
        DIM value$ = ""
        DIM status$ = "ok"
        TRY
            value$ = VALUE$(PAGE$(h, url$), t{"select"}, attr$)
        CATCH

```

```

        status$ = ERRMSG$
    ENDTRY
    DIM row = [now$, t{"name"}, value$, NUMBER$(value$)]
    PUSH(row, url$)
    PUSH(row, status$)
    PUSH(rows, row)
NEXT k
RETURN rows
ENDFUNC

' A line of CSV, quoting the cells that need it.
EXPORT FUNC CSVLINE$(cells)
    RETURN JOIN(SELECT(CsvCell$@, cells), ",")
ENDFUNC

' One cell, in quotes when it holds a comma or a quote.
FUNC CsvCell$(s$)
    DIM q$ = CHR$(34)
    DIM quote = INSTR(s$, ",") >= 0 ORELSE INSTR(s$, q$) >= 0
    IF quote THEN RETURN q$ + REPLACE$(s$, q$, q$ + q$) + q$
    RETURN s$
ENDFUNC

```

How it works

1. NEW sets up a harvester: the name it gives itself in the User-Agent header of every request, the pause between two requests to a site, and a CACHE of pages. The cache is loaded from cache.json if a run left one, and an entry is dropped after cache_hours.
2. PAGE\$ answers a page from the cache while it is fresh. Otherwise RulesFor reads the site's robots.txt once per run and ROBOTS picks the rules that apply: the group of lines that names this harvester, else the group for every visitor (User-agent: *).
3. ALLOWED decides with the longest matching rule, which is how the big search engines read the file: with Disallow: /shop and Allow: /shop/prices, the prices are allowed and the cart is not. MATCHES knows the two wildcards of robots.txt: * for any characters and a final \$ for the end of the address.
4. Pause waits until the last request to the same site is long enough ago: delay_seconds, or the Crawl-delay the site asks for in its robots.txt when that is longer. TICK() gives the time in milliseconds.

5. `GetOnce` sends one request. A busy answer (429 or 5xx) or no answer at all raises an error starting with *transient*, and `RETRY.ATTEMPT` in `Get` tries again after a pause that doubles. Any other answer comes back to `PAGE$`, which keeps a good page and reports a missing one.
6. `VALUE$` parses the page with `HTMLDOM`, finds the first element the selector matches and answers its text with the spaces tidied up, or the attribute.
7. `NUMBER$` turns what the page shows into a plain number. It looks at the last point or comma: followed by exactly three digits and alone, it separates thousands (1,234 pieces); otherwise it is the decimal mark, and every other point or comma is dropped. So 1.234,50 EUR and \$1,234.50 both become 1234.50.
8. `RUN` goes through the targets. A target that fails gets its reason in the status column and the others go on, so one changed shop page does not cost you the whole Monday sheet.

Run it

Try the targets with a dry run first. It reads the pages and shows the values, but writes neither the CSV file nor the cache:

```
jdbasic web_harvester.jdb --dry-run
Copy paper A4, Office Shop: 4,99 € (4.99)
Toner X200, Office Shop: 1.234,50 EUR (1234.50)
2 values read; nothing written
```

A target that failed shows up as a warning with its reason. When the values look right, run it for real; the CSV file is `harvest.csv` in the harvester's folder and the log `harvest.log` next to it. The CSV file starts with the three bytes of a UTF-8 byte order mark, the sign Excel looks for to read the euro sign and other letters right when you open the file with a double click.

Schedule it

The wizard runs the harvester on weekdays at seven. With the default pause of five seconds, fourteen pages from two shops take about a minute. Pages read at seven are kept until seven in the evening, so a second run during the day costs the shops nothing.

Make it yours

The settings are the [web_harvester] part of work.conf:

```
[web_harvester]
folder = "~/Documents/AutomateWork/harvest"
delay_seconds = 5
cache_hours = 12
```

The harvester names itself *AutomateWork harvester (jdBasic)* and adds your mail address from the [user] part, so a site owner who wonders about the requests can write to you.

Changes in the code:

- **One CSV file per week.** In the program, build the file name from the week:

```
DIM week$ = FORMAT_DATE(NOW(), "%Y-W%V")
DIM output$ = PATH.JOIN$(folder$, "harvest-" + week$ + ".csv")
```

- **A mail when a price changes.** Read the last value of each name from the CSV file before RUN, compare it with the new number, and put a message into the outbox with OUTBOX.PUT\$ as the recipes of Chapter 4 do.
- **A longer pause for one site.** If a site asks for more, it says so with Crawl-delay, which the harvester already respects. To be slower on your own, raise delay_seconds.

When it goes wrong

- **"nothing on the page matches":** the shop changed its page and the selector no longer fits. Inspect the page again and correct select.
- **"robots.txt asks to stay out of":** the site does not want this page read by programs. Leave it, or ask the site for another way.
- **"transient: the server answered 503"** after three tries: the site is down or limits requests. The next run tries again.
- **"the page answered 404":** the address is wrong or the product is gone.
- **The euro sign looks wrong in Excel:** the file was started by an older version without the byte order mark. Rename it; the next run starts a new one. In other programs choose UTF-8 when you open it.

Balance dividend

About 45 minutes a week of looking up and typing prices, and a price history that no one has to keep by hand.

X12 Tested Automations

The chore

In March Lena's Monday report covered the wrong week. The clocks had changed on Sunday night, the report counted seven days back from midnight, landed on a Sunday, and reported Sunday to Saturday. Nobody saw it for two weeks, because the numbers looked like numbers. Another time she improved a shared module on a Friday afternoon, and on Monday three recipes stopped with an error that had nothing to do with what she had changed.

Each recipe of this book comes with a test. The tests only help when somebody runs them, and after a change, nobody remembers to run all forty of them.

What you get

Every Sunday evening the test runner looks through the recipes in your work folder, runs every test it finds, each in a process and a folder of its own, and writes a report:

```
Test report 2026-10-04 20:00: 48 tests, 1 failed, 0 flaky

FAILED expert/X09_report_pipeline/report_pipeline_test.jdb
      Error #99: TESTKIT: 2 of 30 assertions failed at line 306
OK easy/E01_downloads_butler/downloads_butler_test.jdb (22 checks)
OK easy/E02_morning_launcher/morning_launcher_test.jdb (16 checks)
```

When a test fails, the report goes to you as a mail in the outbox, so you see it before the Monday report goes out. A test that fails once and passes on the second run is marked *flaky*: it is worth a look, but it does not wake you.

The recipe also brings a test of a different kind: properties of Monday's reports, checked against hundreds of days, amounts and logs that a generator draws at random. They are the check that would have caught the week that started on a Sunday.

Before you start

The recipes live in the `recipes` folder of your work folder, where the wizard copies them, and each has its `_test.jdb` file next to it. The runner starts every

test with the `jdbasic` program the wizard found, the one in the `[paths]` part of `work.conf`. Its own part is short:

```
[tested_automations]
recipes = "~/Documents/AutomateWork/recipes"
reports = "~/Documents/AutomateWork/test reports"
to = ""
```

An empty `to` sends the mail to you, the sender of the `[mail]` part.

The program

The program finds the tests, runs them and writes the report:

```
' Tested Automations: runs the tests of every recipe in your work
' folder on Sunday evening, writes a report and, when a test fails,
' puts a mail to you into the outbox. The settings are the
' [tested_automations] part of work.conf.
'
'   jdbasic test_runner.jdb           run every test
'   jdbasic test_runner.jdb --only M15 the tests of some recipes
'   jdbasic test_runner.jdb --dry-run  list the tests, run nothing
'   jdbasic test_runner.jdb --send     send the outbox

IMPORT WORKCONF, TESTRUN, OUTBOX, MAIL, LOGGER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
    PRINT result{"sent"}; " sent"
END
ENDIF

DIM key$ = "tested_automations."
DIM work$ = "~/Documents/AutomateWork"
DIM root$ = WORKCONF.FOLDER$(cfg, key$ + "recipes", _
    work$ + "/recipes")
DIM reports$ = WORKCONF.FOLDER$(cfg, key$ + "reports", _
    work$ + "/test reports")
DIM to$ = WORKCONF.VALUE(cfg, key$ + "to", "")
IF to$ = "" THEN to$ = OUTBOX.FROM$(cfg)
DIM jdbasic$ = WORKCONF.VALUE(cfg, "paths.jdbasic", "jdbasic")
DIM user_lib$ = PATH.JOIN$(WORKCONF.HOME$(), ".jdbasic", "lib")
root$ = REPLACE$(root$, "\", "/")
IF NOT FILE.ISDIR(root$) THEN THROW "No recipes folder at " + root$
```

```

DIM tests = TESTRUN.FIND(root$)
DIM only$ = WORKCONF.ARG$("only")
IF only$ <> "" THEN
    DIM picked = []
    DIM t = 0
    FOR t = 0 TO LEN(tests) - 1
        DIM file$ = tests[t]
        IF INSTR(file$, only$) >= 0 THEN PUSH(picked, file$)
    NEXT t
    tests = picked
ENDIF
IF WORKCONF.FLAG("dry-run") THEN
    PRINT LEN(tests); " tests below "; root$
    DIM d = 0
    FOR d = 0 TO LEN(tests) - 1
        DIM shown$ = tests[d]
        PRINT " "; shown$
    NEXT d
    END
ENDIF

MKDIR reports$
DIM lg = LOGGER.NEW("tested_automations")
DIM log$ = PATH.JOIN$(reports$, "test_runner.log")
LOGGER.TO_FILE(lg, log$, 1000000, 3)
LOGGER.INFO(lg, "run started", {"tests": LEN(tests)})
DIM first = []
DIM given$ = GETENV$("JDBASIC_PATH")
IF given$ <> "" THEN first = SPLIT(given$, ";")
PUSH(first, user_lib$)
DIM search$ = TESTRUN.SEARCH_PATH$(root$, first)
DIM results = TESTRUN.RUN_ALL(jdbasic$, tests, search$)
SETENV "JDBASIC_PATH", given$

DIM now$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M")
DIM report$ = TESTRUN.REPORT$(results, now$)
DIM today$ = LEFT$(now$, 10)
DIM report_file$ = PATH.JOIN$(reports$, "tests " + today$ + ".txt")
TXTWRITER report_file$, report$
PRINT report$
DIM bad = TESTRUN.TALLY_OF(results, "failed")
LOGGER.INFO(lg, "run done", {"failed": bad, "report": report_file$})
IF bad > 0 AND ALSO to$ <> "" THEN
    DIM subject$ = STR$(bad) + " recipe tests failed"
    DIM msg = MAIL.MESSAGE(OUTBOX.FROM$(cfg), to$, subject$)
    MAIL.PLAIN(msg, report$)
    DIM eml$ = OUTBOX.PUT$(msg, cfg, "tests " + today$ + ".eml")
    LOGGER.WARN(lg, "mail in the outbox", {"path": eml$})

```

```

    PRINT "In the outbox: "; eml$
ENDIF
IF bad > 0 THEN THROW STR$(bad) + " tests failed, see " + report_file$

```

The module does the finding and the running:

```

' testrun.jdb: runs the tests of all your recipes, each in a process
' of its own, and writes what happened as a report. A test is a file
' that ends in _test.jdb or _props.jdb. Every recipe folder is on the
' module search path of the tests, so a test can import the module of
' another recipe.
'
'   IMPORT TESTRUN
'   DIM tests = TESTRUN.FIND("C:/Users/lena/AutomateWork/recipes")
'   DIM results = TESTRUN.RUN_ALL(jdbasic$, tests, search$)
'   PRINT TESTRUN.REPORT$(results, "2026-10-04 20:00")

EXPORT MODULE TESTRUN

' The folders below root$, root$ itself first, three levels deep.
EXPORT FUNC FOLDERS(root$)
    DIM out = [root$]
    DIM k = 0
    DO WHILE k < LEN(out)
        DIM here$ = out[k]
        DIM rest$ = MID$(here$, LEN(root$), LEN(here$))
        DIM depth = LEN(SPLIT(rest$, "/"))
        IF depth <= 3 THEN
            DIM names = SORT(DIR$(PATH.JOIN$(here$, "*")))
            DIM j = 0
            FOR j = 0 TO LEN(names) - 1
                DIM name$ = names[j]
                DIM sub$ = here$ + "/" + name$
                DIM hidden = LEFT$(name$, 1) = "."
                IF FILE.ISDIR(sub$) AND ALSO NOT hidden THEN
                    PUSH(out, sub$)
                ENDIF
            NEXT j
        ENDIF
        k = k + 1
    LOOP
    RETURN out
ENDFUNC

' The test files below root$, sorted by path.
EXPORT FUNC FIND(root$)
    DIM dirs = FOLDERS(root$)

```

```

    DIM out = []
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(dirs) - 1
        DIM dir$ = dirs[k]
        DIM names = DIR$(PATH.JOIN$(dir$, "*.jdb"))
        FOR j = 0 TO LEN(names) - 1
            DIM name$ = names[j]
            IF ENDSWITH(name$, "_test.jdb") ORELSE _
                ENDSWITH(name$, "_props.jdb") THEN
                PUSH(out, dir$ + "/" + name$)
            ENDIF
        NEXT j
    NEXT k
    RETURN SORT(out)
ENDFUNC

' A folder or file as a full path with forward slashes; a path that
' does not exist is answered unchanged.
EXPORT FUNC ABSOLUTE$(path$)
    DIM p$ = REPLACE$(path$, "\", "/")
    IF MID$(p$, 1, 1) = ":" ORELSE LEFT$(p$, 1) = "/" THEN RETURN p$
    IF NOT FILE.EXISTS(p$) THEN RETURN p$
    DIM dir$ = p$
    DIM name$ = ""
    IF NOT FILE.ISDIR(p$) THEN
        dir$ = PATH.DIRNAME$(p$)
        name$ = PATH.BASENAME$(p$)
        IF dir$ = "" THEN dir$ = "."
    ENDIF
    DIM back$ = CD()
    DIM full$ = REPLACE$(CD(dir$), "\", "/")
    DIM moved$ = CD(back$)
    IF name$ = "" THEN RETURN full$
    RETURN full$ + "/" + name$
ENDFUNC

' The module search path for the tests, as full paths: the folders
' given first, then every folder below root$ that holds a .jdb file.
EXPORT FUNC SEARCH_PATH$(root$, first)
    DIM parts = []
    DIM k = 0
    FOR k = 0 TO LEN(first) - 1
        DIM given$ = first[k]
        PUSH(parts, ABSOLUTE$(given$))
    NEXT k
    DIM dirs = FOLDERS(root$)
    FOR k = 0 TO LEN(dirs) - 1

```

```

        DIM dir$ = dirs[k]
        DIM sources = DIR$(PATH.JOIN$(dir$, "*.jdb"))
        IF LEN(sources) > 0 THEN PUSH(parts, dir$)
    NEXT k
    RETURN JOIN(parts, ";")
ENDFUNC

' Text without the colour codes of a terminal.
EXPORT FUNC PLAIN$(text$)
    DIM esc$ = CHR$(27)
    DIM out$ = ""
    DIM i = 0
    DO WHILE i < LEN(text$)
        DIM c$ = MID$(text$, i, 1)
        IF c$ = esc$ THEN
            DO WHILE i < LEN(text$) AND ALSO MID$(text$, i, 1) <> "m"
                i = i + 1
            LOOP
        ELSE
            out$ = out$ + c$
        ENDIF
        i = i + 1
    LOOP
    RETURN out$
ENDFUNC

' The counts of a TESTKIT summary line ("12 passed, 1 failed"), as
' a map with "passed" and "failed"; both -1 when there is none.
EXPORT FUNC COUNTS(output$)
    DIM out = {"passed": -1, "failed": -1}
    DIM text$ = REPLACE$(PLAIN$(output$), CHR$(13), "")
    DIM lines = SPLIT(text$, CHR$(10))
    DIM k = 0
    FOR k = 0 TO LEN(lines) - 1
        DIM words = SPLIT(TRIM$(lines[k]), " ")
        IF LEN(words) >= 4 AND ALSO words[1] = "passed," THEN
            out{"passed"} = VAL(words[0])
            out{"failed"} = VAL(words[2])
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

FUNC LastLine$(output$)
    DIM text$ = TRIM$(REPLACE$(PLAIN$(output$), CHR$(13), ""))
    DIM lines = SPLIT(text$, CHR$(10))
    IF LEN(lines) = 0 THEN RETURN ""
    RETURN lines[LEN(lines) - 1]

```

```
ENDFUNC
```

```
FUNC RunOnce(jdbasic$, file$)
    DIM t0 = TICK()
    DIM program$ = jdbasic$
    IF INSTR(program$, "/") >= 0 ORELSE INSTR(program$, "\" ) >= 0 THEN
        program$ = ABSOLUTE$(program$)
    ENDIF
    DIM full$ = ABSOLUTE$(file$)
    DIM back$ = CD()
    DIM moved$ = CD(PATH.DIRNAME$(full$))
    DIM r = OS.EXEC(program$, [PATH.BASENAME$(full$)])
    moved$ = CD(back$)
    DIM output$ = r{"OUTPUT"}
    DIM c = COUNTS(output$)
    DIM out = {"file": file$, "exit": r{"EXIT_CODE"}}
    out{"ok"} = r{"EXIT_CODE"} = 0 ANDALSO c{"failed"} <= 0
    out{"passed"} = c{"passed"}
    out{"failed"} = c{"failed"}
    out{"ms"} = TICK() - t0
    out{"last"} = LastLine$(output$)
    RETURN out
ENDFUNC
```

```
' Runs one test in its own folder with the module search path
' search$. A test that fails is run a second time: "status" is "ok",
' "flaky" when only the second run passed, or "failed".
```

```
EXPORT FUNC RUN_ONE(jdbasic$, file$, search$)
    DIM parts = SPLIT(search$, ";")
    DIM full = []
    DIM k = 0
    FOR k = 0 TO LEN(parts) - 1
        DIM part$ = parts[k]
        IF part$ <> "" THEN PUSH(full, ABSOLUTE$(part$))
    NEXT k
    SETENV "JDBASIC_PATH", JOIN(full, ";")
    DIM first = RunOnce(jdbasic$, file$)
    IF first{"ok"} THEN
        first{"status"} = "ok"
        RETURN first
    ENDIF
    DIM again = RunOnce(jdbasic$, file$)
    IF again{"ok"} THEN
        again{"status"} = "flaky"
        RETURN again
    ENDIF
    first{"status"} = "failed"
    RETURN first
```


ENDFUNC

```
' Runs every test of the list and answers one result map per test.
```

```
EXPORT FUNC RUN_ALL(jdbasic$, files, search$)
```

```
DIM out = []
```

DIM k = 0

```
FOR k = 0 TO LEN(files) - 1
```

```
DIM file$ = files[k]
```

```
PUSH(out, RUN_ONE(jdbasic$, file$, search$))
```

NEXT k

RETURN out

ENDFUNC

```
' How many results have the status, "ok", "flaky" or "failed".
```

```
EXPORT_FUNC TALLY_OF(results, status$)
```

```
DIM n = 0
```

DIM k = 0

```
FOR k = 0 TO LEN(results) - 1
```

```
DIM r = results[k]
```

```
IF r{"status"} = status$ THEN n = n + 1
```

NEXT k

RETURN n

ENDFUNC

```
' The results as a report: a summary line, then one line per test,
```

```
' the failed ones first with the last line they printed.
```

```
EXPORT FUNC REPORT$(results, when$)
```

```
DIM LF$ = CHR$(10)
```

```
DIM bad = TALLY_OF(results, "failed")
```

```
DIM out$ = "Test report " + when$ + ": " + STR$(LEN(results))
```

```
out$ = out$ + " tests, " + STR$(bad) + " failed, "
```

```
DIM flaky = TALLY_OF(results, "flaky")
```

```
out$ = out$ + STR$(flaky) + " flaky" + LF$ + LF$
```

```
DIM order = ["failed", "flaky", "ok"]
```

```
DIM S = 0
```

DIM k = 0

FOR s = 0 TO 2

```
FOR k = 0 TO LEN(results) - 1
```

```
DIM r = results[k]
```

```
IF r{"status"} = order[s] THEN
```

```
DIM line$ = UCASE$(order[s]) + " " + r{"file"}
```

```
IF r{"passed"} >= 0 THEN
```

```
DIM n$ = STR$(r{"passed"})
```

```
line$ = line$ + " (" + n$ + " checks)"
```

ENDIF

```
out$ = out$ + line$ + LF$
```

```
IF order[s] <> "ok" THEN
```

```
out$ = out$ + "      " + r{"last"} + LF$
```

```

        ENDIF
    ENDIF
NEXT k
NEXT s
RETURN out$
ENDFUNC

```

And these are the properties of Monday's reports, in the folder props of the recipe:

```

' Properties of Monday's reports: the week of the Report Pipeline
' (X09) and the weeks of the Timesheet for the Boss (M15), checked
' against hundreds of drawn days, amounts and logs. The test runner
' puts both recipe folders on the module search path.

IMPORT TESTKIT, PROPTEST, PIPELINE, TIMESHEET, DF

DIM gStart = CDATE("2026-01-01 12:00:00")
DIM gCenters = ["Sales", "Marketing", "IT", "Travel"]
DIM gSeed = {"seed": 20261005, "runs": 200}

FUNC Day$(offset)
    RETURN FORMAT_DATE(DATEADD("D", offset, gStart), "%Y-%m-%d")
ENDFUNC

FUNC Noon(day$)
    RETURN CDATE(day$ + " 12:00:00")
ENDFUNC

FUNC DayOfWeek(day$)
    RETURN VAL(FORMAT_DATE(Noon(day$), "%w"))
ENDFUNC

FUNC WeekIsBefore(offset)
    DIM day$ = Day$(offset)
    DIM week = PIPELINE.LAST_WEEK(day$)
    IF DayOfWeek(week{"from"}) <> 1 THEN RETURN FALSE
    DIM span = DATEDIFF("D", Noon(week{"from"}), Noon(week{"to"}))
    DIM gap = DATEDIFF("D", Noon(week{"to"}), Noon(day$))
    RETURN span = 6 AND ALSO gap >= 1 AND ALSO gap <= 7
ENDFUNC

FUNC RecipesAgree(offset)
    DIM day$ = Day$(offset)
    DIM week = PIPELINE.LAST_WEEK(day$)
    DIM next_monday = DATEADD("D", 7, Noon(week{"from"}))
    DIM m15$ = TIMESHEET.MONDAY$(day$)

```

```

    RETURN FORMAT_DATE(next_monday, "%Y-%m-%d") = m15$
ENDFUNC

FUNC MoneyReadsBack(cents)
    DIM text$ = PIPELINE.MONEY$(cents / 100)
    DIM back = VAL(REPLACE$(text$, ",", ""))
    RETURN ABS(back - cents / 100) < 0.001
ENDFUNC

FUNC NothingLost(amounts)
    DIM rows = []
    DIM k = 0
    FOR k = 0 TO LEN(amounts) - 1
        DIM cc$ = gCenters[k MOD 4]
        PUSH(rows, {"day": "2026-10-01", "cost_center": cc$, _
            "amount": amounts[k]})
    NEXT k
    DIM budgets = {"Sales": 4000, "IT": 3000}
    DIM report = PIPELINE.REPORT(rows, budgets)
    RETURN SUM(DF.COL(report, "spent")) = SUM(amounts)
ENDFUNC

FUNC MonthAddsUp(quarters)
    DIM sessions = []
    DIM k = 0
    FOR k = 0 TO LEN(quarters) - 1
        DIM start = DATEADD("D", k, CDATE("2026-09-25 09:00:00"))
        DIM finish = DATEADD("N", 15 * quarters[k], start)
        DIM s$ = FORMAT_DATE(start, "%Y-%m-%d %H:%M:%S")
        DIM e$ = FORMAT_DATE(finish, "%Y-%m-%d %H:%M:%S")
        PUSH(sessions, {"project": gCenters[k MOD 4], "start": s$, _
            "stop": e$})
    NEXT k
    DIM now$ = "2026-11-02 08:00:00"
    DIM totals = TIMESHEET.MONTH_TOTALS(sessions, "2026-10", now$)
    DIM month_sum = 0.0
    DIM names = MAP.KEYS(totals)
    FOR k = 0 TO LEN(names) - 1
        month_sum = month_sum + totals{names[k]}
    NEXT k
    DIM week_sum = 0.0
    DIM mondays = TIMESHEET.WEEKS("2026-10")
    DIM w = 0
    FOR w = 0 TO LEN(mondays) - 1
        DIM table = TIMESHEET.WEEK_HOURS(sessions, mondays[w], _
            "2026-10", now$)
        DIM projects = MAP.KEYS(table)
        FOR k = 0 TO LEN(projects) - 1

```

```

        week_sum = week_sum + SUM(table{projects[k]})
    NEXT k
NEXT w
RETURN ABS(month_sum - week_sum) < 0.001
ENDFUNC

TESTKIT.SUITE("Monday's report (X09)")
PROPTTEST.FORALL("the week reported is the week before", _
    "int:0..1500", WeekIsBefore@, gSeed)
PROPTTEST.FORALL("an amount reads back as written", _
    "int:-100000000..100000000", MoneyReadsBack@, gSeed)
PROPTTEST.FORALL("no booking is lost on the way to the report", _
    "list(int:0..5000,0..40)", NothingLost@, gSeed)

TESTKIT.SUITE("two recipes, one calendar (X09 and M15)")
PROPTTEST.FORALL("X09 and M15 agree on the Monday", _
    "int:0..1500", RecipesAgree@, gSeed)
PROPTTEST.FORALL("the weeks of a timesheet add up to its month", _
    "list(int:0..40,0..45)", MonthAddsUp@, gSeed)

TESTKIT.REPORT()

```

How it works

1. **Finding the tests.** FOLDERS walks the recipes folder three levels deep and leaves out folders whose name starts with a dot. FIND keeps every file that ends in `_test.jdb` or `_props.jdb`.
2. **One search path for all.** A test imports the module of its recipe, and a property test may import the modules of other recipes as well. `SEARCH_PATH$` puts every folder that holds a `.jdb` file on the module search path, after the folders that were there already, and `RUN_ONE` sets it as `JDBASIC_PATH` for the test.
3. **A process and a folder of its own.** `RunOnce` changes into the folder of the test with `CD`, starts `jdbasic` on it with `OS.EXEC` and changes back. A test that crashes takes only its own process down, and a test that reads a sample file next to itself finds it. Every path is made absolute with `ABSOLUTE$` first, because after the change of folder a relative path points somewhere else.
4. **Reading the result.** `TESTKIT` ends with a line like *22 passed, 0 failed*. `PLAIN$` takes out the colour codes of the console, and `COUNTS` reads the two numbers. A test is fine when its process ended with 0 and no check failed.

5. **Flaky or failed.** A test that fails is run once more. When the second run passes, the result says `flaky`; a test that depends on the clock or on a port that was taken behaves like that.
6. **The report.** `REPORT$` writes the failed tests first, each with the last line it printed, then the flaky ones, then the rest. The program writes it to a file per day, prints it, puts it into a mail when something failed, and ends with an error itself, so the Task Scheduler or the X01 job server marks the run as failed too.
7. **Properties.** A property is a function that answers `TRUE` or `FALSE` for one input. `PROPTEST.FORALL` draws 200 inputs from a generator, `"int:0..1500"` for a whole number in that range, or `"list(int:0..5000,0..40)"` for a list of up to 40 amounts, and calls the function with each. `WeekIsBefore` turns a number into a day of the years 2026 to 2030 and checks that the week X09 reports starts on a Monday, ends six days later and lies before the day. `RecipesAgree` checks that X09 and the timesheet of M15 agree on the Monday of a week. The seed in `gSeed` makes every run draw the same inputs, so a failure can be repeated.

When a property fails, `PROPTEST` looks for the smallest input that still fails and names it with the seed. This is what the properties said about the version of X09 that counted from midnight:

```
Monday's report (X09)
FAIL 1  the week reported is the week before: falsified by 802
        (seed 20261005, case 3, shrunk from 804)
ok  2  an amount reads back as written (200 cases)
ok  3  no booking is lost on the way to the report (200 cases)

two recipes, one calendar (X09 and M15)
FAIL 4  X09 and M15 agree on the Monday: falsified by 802
        (seed 20261005, case 3, shrunk from 804)
ok  5  the weeks of a timesheet add up to its month (200 cases)
```

Day 802 after 1 January 2026 is Monday 13 March 2028, the first Monday after the clocks change in the United States. Within a second the test names a day that a person would have had to wait two years for.

Run it

With the long lines wrapped:

```
jdbasic test_runner.jdb --dry-run
48 tests below C:/Users/lena/Documents/AutomateWork/recipes
  C:/Users/lena/Documents/AutomateWork/recipes/easy/
    E01_downloads_butler/downloads_butler_test.jdb
  ...

jdbasic test_runner.jdb --only M15
Test report 2026-10-02 06:44: 1 tests, 0 failed, 0 flaky

OK  C:/Users/lena/Documents/AutomateWork/recipes/medium/
    M15_timesheet/timesheet_test.jdb  (31 checks)
```

All 48 tests of this book take about half a minute. `--only` takes any part of a path, so `--only expert` runs the tests of one level and `--only X09` those of one recipe.

Schedule it

The wizard plans the runner for Sunday at 20:00, so the report is there before the first recipe of the week starts. In the X01 job server it is one more job:

```
[[job]]
name = "tests"
when = "weekly sun 20:00"
program = "X12_tested_automations/test_runner.jdb"
```

Run it by hand as well, every time you have changed a recipe or a shared module. That is the moment it pays most.

Make it yours

- **A property of your own.** Add a function to `monday_props.jdb` and one `FORALL` line. This one checks that every month of the timesheet has four to six weeks:

```
FUNC FourToSixWeeks(offset)
  DIM month$ = LEFT$(Day$(offset), 7)
  DIM n = LEN(TIMESHEET.WEEKS(month$))
  RETURN n >= 4 AND ALSO n <= 6
```

```
ENDFUNC
```

```
PROPTTEST.FORALL("a month has four to six weeks", _
    "int:0..1500", FourToSixWeeks@, gSeed)
```

- **Search harder on the weekend.** Raise runs in gSeed to 2000. The properties then take a few seconds instead of a fraction of one.
- **A properties file for another recipe.** Any file that ends in `_props.jdb` below the recipes folder is run, and it may import the module of any recipe. Keep such files in a folder of their own, such as `props`, as this recipe does, and check them through the runner: `jdbbasic --lint` on its own does not know where the other recipes keep their modules.

When it goes wrong

- **"No recipes folder at":** set recipes to the folder the wizard copied the recipes to.
- **Every test fails with "cannot load module":** the module search path misses the libraries. Start the runner from a command prompt where `jdbbasic` itself finds its libraries, or add their folder to `JDBASIC_PATH` before the runner starts.
- **A test passes alone and fails in the runner:** it expects to be started from a particular folder, or it uses a port or a file that another test also uses. The runner starts every test in its own folder; a test should find its samples there and make its own temporary folders with `MKTEMP$`.
- **The runner does not finish:** a test waits for input or for a server that does not answer. The runner has no time limit; find the test with `--only` and the levels, and give it one.

Balance dividend

About 30 minutes a week: the reports that would have gone out wrong, the Monday mornings spent finding out why a recipe stopped, and the courage to change a shared module on a Friday.

X13 Health Checks and Alerts

The chore

The job server runs Lena's jobs and writes down what happened. Nobody reads it. She learns that the Friday backup failed when she needs the backup, and that the server was not running when a colleague asks on Monday where the report is. The other way round is just as tiring: a check that mails her every fifteen minutes about the same failed job gets filtered away within a week. What she wants is one message when something goes wrong, one when it is over, and silence in between.

What you get

Every fifteen minutes the health check looks at the job server and the outbox. It finds four kinds of trouble:

Problem	When it counts
The server	no lock file, or no sign of life for 10 minutes
A failed job	its last run ended with an error
A late job	it was due more than 30 minutes ago and has not run
A waiting mail	it has sat in the outbox for more than a day

Each new problem is announced once, in a window on the screen and in a mail. When it goes away, a second mail says so. Each run also writes the numbers of every job into a file that a monitoring system can read.

```
jdbasic health_check.jdb --dry-run
4 jobs, 2 problems
  backup failed at 2026-10-09 16:31: exit 1: the drive E: is gone
  invoice 0413.eml has waited in the outbox for 27 hours.
would alert lena@example.com
```

Before you start

The job server (X01) runs, and the [mail] part of `work.conf` holds your mail server, as in Chapter 4. Without a jobs file the check still watches the outbox.

The alerts go into the outbox like every other mail of this book, and you send them with the `--send` of any medium recipe. To have them go out at once, store the mail password with X14 and switch on `send_alerts`. The check then sends without asking you first, so it sends only to the address in `alert_to`, which is your own unless you change it.

The program

The main program gathers what is on disk, lets the module find the problems, and sends what is new:

```
' Health Checks and Alerts: looks at the job server and the outbox,
' tells you about each new problem once, and again when it is over.
' The alert goes into the outbox, or straight out when the mail
' password is in the Credential Manager (X14) and send_alerts is on.
' The settings are the [health_checks] part of work.conf.
'
'   jdbasic health_check.jdb          check, alert, keep the numbers
'   jdbasic health_check.jdb --dry-run  check and show only

IMPORT WORKCONF, JOBSTATE, HEALTH, OUTBOX, VAULT, DT

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "health_checks."
DIM self$ = WORKCONF.VALUE(cfg, "user.email", "")
DIM alert_to$ = WORKCONF.VALUE(cfg, key$ + "alert_to", self$)
IF alert_to$ = "" THEN alert_to$ = self$
DIM send_alerts = WORKCONF.VALUE(cfg, key$ + "send_alerts", FALSE)
DIM notify$ = WORKCONF.VALUE(cfg, key$ + "notify", "window")
DIM dry_run = WORKCONF.FLAG("dry-run")
DIM folder$ = JOBSTATE.FOLDER$(cfg)
DIM alerts_path$ = PATH.JOIN$(folder$, "alerts.json")
DIM now = DT.NOW()
DIM offset = DT.LOCAL_OFFSET()

DIM jobs = []
DIM jobs_path$ = JOBSTATE.JOBS_PATH$(cfg)
IF FILE.EXISTS(jobs_path$) THEN jobs = JOBSTATE.JOBS(jobs_path$)
DIM state = JOBSTATE.READ(JOBSTATE.STATE_PATH$(cfg))
DIM beat = JOBSTATE.HEARTBEAT_AGE(JOBSTATE.LOCK_PATH$(cfg), now)

DIM box = []
DIM pending = OUTBOX.PENDING(cfg)
DIM k = 0
FOR k = 0 TO LEN(pending) - 1
    DIM file$ = pending[k]
```

```

    DIM info = FILE.STAT(file$)
    DIM mins = DATEDIFF("n", CDATE(info{"mtime"}), NOW())
    PUSH(box, {"name": PATH.BASENAME$(file$), "hours": mins / 60})
NEXT k

DIM facts AS MAP
facts{"heartbeat"} = beat
DIM server_min = WORKCONF.VALUE(cfg, key$ + "server_minutes", 10)
facts{"server_minutes"} = server_min
facts{"late_minutes"} = WORKCONF.VALUE(cfg, key$ + "late_minutes", 30)
facts{"outbox"} = box
facts{"outbox_hours"} = WORKCONF.VALUE(cfg, key$ + "outbox_hours", 24)
DIM problems = HEALTH.CHECK(jobs, state, now, offset, facts)

DIM alerts = JOBSTATE.READ(alerts_path$)
DIM change = HEALTH.DIFF(alerts, problems, now)
DIM fresh = change{"new"}
DIM gone = change{"resolved"}

DIM count$ = STR$(LEN(problems)) + " problems"
IF LEN(problems) = 1 THEN count$ = "1 problem"
PRINT LEN(jobs); " jobs, "; count$
FOR k = 0 TO LEN(problems) - 1
    DIM p = problems[k]
    PRINT " "; p{"text"}
NEXT k
IF dry_run THEN
    IF LEN(fresh) + LEN(gone) > 0 THEN PRINT "would alert "; alert_to$
    END
ENDIF

MKDIR folder$
DIM prom$ = HEALTH.METRICS_TEXT$(jobs, state, problems, beat)
TXTWRITER PATH.JOIN$(folder$, "metrics.prom"), prom$
TXTWRITER alerts_path$, JSON.STRINGIFY$(change{"open"})
IF LEN(fresh) + LEN(gone) = 0 THEN END

DIM msg = HEALTH.MESSAGE(OUTBOX.FROM$(cfg), alert_to$, change)
DIM sent = FALSE
IF send_alerts THEN
    DIM env$ = "AUTOMATEWORK_MAIL"
    DIM password$ = VAULT.FIND$("AutomateWork/mail", env$, "")
    IF password$ = "" THEN
        PRINT "No mail password in the Credential Manager."
    ELSE
        DIM server = OUTBOX.SERVER(cfg, password$)
        DIM rep = HEALTH.SEND(msg, server, 3, 5000)
        sent = rep{"ok"}
    END IF
END IF

```

```

        IF sent THEN PRINT "Alert sent to "; alert_to$
        IF NOT sent THEN PRINT "Not sent: "; rep{"error"}
    ENDIF
ENDIF
IF NOT sent THEN
    DIM stamp$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H%M%S")
    DIM eml$ = "alert " + stamp$ + ".eml"
    DIM put$ = OUTBOX.PUT$(msg, cfg, eml$)
    PRINT "In the outbox: "; eml$
ENDIF
IF notify$ = "window" AND ALSO LEN(fresh) > 0 THEN
    DIM button = MSGBOX(HEALTH.TEXT$(change), 48, "Health check")
ENDIF

```

The module holds the checks, the memory of what was already announced, the mail, the numbers and the sending:

```

' health.jdb: the checks of the health check. It looks at what the job
' server (X01) wrote and at the outbox, finds what is wrong, tells new
' problems from known ones, and writes the numbers for a monitoring
' system.
'
'   IMPORT HEALTH
'   DIM problems = HEALTH.CHECK(jobs, state, now, offset, facts)
'   DIM change = HEALTH.DIFF(alerts, problems, now)
'   IF LEN(change{"new"}) > 0 THEN PRINT HEALTH.TEXT$(change)
'
' A problem is a map with "key", the same for the same trouble from
' one run to the next, "kind" and "text".

EXPORT MODULE HEALTH

IMPORT SCHED, METRICS, MAIL, RETRY, DT

' Local time text of a moment.
FUNC Clock$(t, offset)
    RETURN LEFT$(REPLACE$(DT.ISO$(t, offset), "T", " "), 16)
ENDFUNC

SUB Add(out, key$, kind$, text$)
    DIM p AS MAP
    p{"key"} = key$
    p{"kind"} = kind$
    p{"text"} = text$
    PUSH(out, p)
ENDSUB

```

```

' A server that never marked its lock (beat -1) or not for too long.
SUB ServerProblems(out, beat, minutes)
  IF beat < 0 THEN
    Add(out, "server", "server", "The job server is not running.")
  ELSEIF beat > minutes * 60 THEN
    DIM m$ = STR$(INT(beat / 60))
    DIM t$ = "The job server has not signed for " + m$
    Add(out, "server", "server", t$ + " minutes.")
  ENDIF
ENDSUB

' A job whose last run failed, or that missed a run time by more than
' late_s seconds.
SUB JobProblems(out, job, rec, now, offset, late_s)
  DIM name$ = job{"name"}
  IF MAP.EXISTS(rec, "last_ok") THEN
    IF NOT rec{"last_ok"} THEN
      DIM ended$ = rec{"last_end"}
      DIM why$ = rec{"last_error"}
      DIM t$ = name$ + " failed at " + ended$ + ": " + why$
      Add(out, "failed:" + name$, "failed", t$)
    ENDIF
  ENDIF
  DIM expected = SCHED.PREVRUN(job{"when"}, now - late_s, offset)
  IF rec{"last_due"} < expected THEN
    DIM at$ = Clock$(expected, offset)
    DIM late$ = name$ + " was due at " + at$ + " and has not run."
    Add(out, "late:" + name$, "late", late$)
  ENDIF
ENDSUB

' The problems at now. facts holds what the caller found on disk:
' "heartbeat" (the age of the server's lock mark in seconds, -1 for
' none), "server_minutes", "late_minutes", "outbox" (a list of maps
' with "name" and "hours") and "outbox_hours".
EXPORT FUNC CHECK(jobs, state, now, offset, facts)
  DIM out = []
  IF LEN(jobs) > 0 THEN
    DIM beat = facts{"heartbeat"}
    ServerProblems(out, beat, facts{"server_minutes"})
  ENDIF
  DIM late_s = facts{"late_minutes"} * 60
  DIM k = 0
  FOR k = 0 TO LEN(jobs) - 1
    DIM job = jobs[k]
    DIM name$ = job{"name"}
    IF job{"enabled"} AND ALSO MAP.EXISTS(state, name$) THEN
      JobProblems(out, job, state{name$}, now, offset, late_s)
    ENDIF
  NEXT k

```

```

        ENDIF
    NEXT k
    DIM box = facts{"outbox"}
    FOR k = 0 TO LEN(box) - 1
        DIM item = box[k]
        IF item{"hours"} > facts{"outbox_hours"} THEN
            DIM file$ = item{"name"}
            DIM h$ = STR$(INT(item{"hours"}))
            DIM t$ = file$ + " has waited in the outbox for " + h$
            Add(out, "outbox:" + file$, "outbox", t$ + " hours.")
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' What changed against the open alerts of the last run, a map from
' key to the time the alert was first seen and its text: "new"
' problems, "resolved" ones as maps with key and text, and "open",
' the alerts to keep for the next run.
EXPORT FUNC DIFF(alerts, problems, now)
    DIM open AS MAP
    DIM fresh = []
    DIM k = 0
    FOR k = 0 TO LEN(problems) - 1
        DIM p = problems[k]
        DIM key$ = p{"key"}
        DIM entry AS MAP
        entry{"text"} = p{"text"}
        IF MAP.EXISTS(alerts, key$) THEN
            entry{"since"} = alerts{key$}{"since"}
        ELSE
            entry{"since"} = now
            PUSH(fresh, p)
        ENDIF
        open{key$} = entry
    NEXT k
    DIM over = FILTER(LAMBDA USE(open) k$ -> _
        NOT MAP.EXISTS(open, k$), MAP.KEYS(alerts))
    DIM gone = SELECT(LAMBDA USE(alerts) k$ -> _
        {"key": k$, "text": alerts{k$}{"text"}}, over)
    DIM out AS MAP
    out{"new"} = fresh
    out{"resolved"} = gone
    out{"open"} = open
    RETURN out
ENDFUNC

' The alert as text: new problems, then the ones that are over.

```

```

EXPORT FUNC TEXT$(change)
    DIM lines = []
    DIM fresh = change{"new"}
    DIM gone = change{"resolved"}
    IF LEN(fresh) > 0 THEN PUSH(lines, "New problems:")
    lines = APPEND(lines, SELECT(LAMBDA p -> " " + p{"text"}, fresh))
    IF LEN(gone) > 0 THEN PUSH(lines, "Over now:")
    lines = APPEND(lines, SELECT(LAMBDA g -> " " + g{"text"}, gone))
    PUSH(lines, "Still open: " + STR$(LEN(MAP.KEYS(change{"open"}))))
    RETURN JOIN(lines, CHR$(10))
ENDFUNC

' The alert mail.
EXPORT FUNC MESSAGE(from$, to$, change)
    DIM n = LEN(change{"new"})
    DIM subject$ = "AutomateWork: all clear"
    IF n = 1 THEN subject$ = "AutomateWork: 1 new problem"
    DIM n$ = STR$(n)
    IF n > 1 THEN subject$ = "AutomateWork: " + n$ + " new problems"
    DIM msg = MAIL.MESSAGE(from$, to$, subject$)
    MAIL.PLAIN(msg, TEXT$(change))
    RETURN msg
ENDFUNC

FUNC Gauge(name$, doc$, by_job)
    DIM full$ = "automatework_" + name$
    IF by_job THEN RETURN METRICS.NEWGAUGE(full$, doc$, ["job"])
    RETURN METRICS.NEWGAUGE(full$, doc$)
ENDFUNC

' The state of the jobs in the text format of Prometheus, for a
' monitoring system that reads files.
EXPORT FUNC METRICS_TEXT$(jobs, state, problems, heartbeat)
    METRICS.RESET()
    DIM ok_g = Gauge("job_ok", "1 when the last run worked.", TRUE)
    DIM runs_g = Gauge("job_runs", "Runs of the job.", TRUE)
    DIM fails_g = Gauge("job_fails", "Failed runs of the job.", TRUE)
    DIM won$ = "job_last_success_seconds"
    DIM won_g = Gauge(won$, "Last run that worked.", TRUE)
    DIM beat$ = "server_heartbeat_age_seconds"
    DIM beat_g = Gauge(beat$, "Age of the lock mark.", FALSE)
    DIM prob_g = Gauge("problems", "Problems found.", FALSE)
    DIM k = 0
    FOR k = 0 TO LEN(jobs) - 1
        DIM job = jobs[k]
        DIM name$ = job{"name"}
        IF MAP.EXISTS(state, name$) THEN
            DIM rec = state{name$}

```

```

        DIM ok = 1
        IF MAP.EXISTS(rec, "last_ok") THEN
            IF NOT rec{"last_ok"} THEN ok = 0
        ENDIF
        METRICS.SETVALUE(ok_g, ok, [name$])
        METRICS.SETVALUE(runs_g, rec{"runs"}, [name$])
        METRICS.SETVALUE(fails_g, rec{"fails"}, [name$])
        METRICS.SETVALUE(won_g, rec{"last_success"}, [name$])
    ENDIF
NEXT k
METRICS.SETVALUE(beat_g, heartbeat)
METRICS.SETVALUE(prob_g, LEN(problems))
RETURN METRICS.EXPOSE$()
ENDFUNC

' Sends one mail; a failed delivery is an error.
FUNC SendOnce(job)
    DIM r = MAIL.SEND(job{"msg"}, job{"server"})
    DIM said$ = TRIM$(r{"output"})
    DIM code$ = STR$(r{"exit_code"})
    IF NOT r{"ok"} THEN THROW "curl " + code$ + ": " + said$
    RETURN TRUE
ENDFUNC

' Sends the alert with retries; answers the report of RETRY.ATTEMPT.
EXPORT FUNC SEND(msg, server, attempts, delay_ms)
    DIM job AS MAP
    job{"msg"} = msg
    job{"server"} = server
    DIM opts AS MAP
    opts{"attempts"} = attempts
    opts{"delay"} = delay_ms
    RETURN RETRY.ATTEMPT(SendOnce@, job, opts)
ENDFUNC

```

How it works

1. **The facts.** The main program reads the jobs and the state file of the job server through `JOBSTATE`, the age of the server's lock mark, and the age of every mail in the outbox. It passes them to `HEALTH.CHECK` as plain values, so the test can hand in any situation it likes.
2. **The server.** `ServerProblems` complains when there is no lock file or when the server has not marked it for `server_minutes`. Without any jobs the server is not needed, and there is no complaint.

3. **Failed and late jobs.** JobProblems looks at the last result of each job. For lateness it asks SCHED for the last run time before *now minus late_minutes*. If the job's last run was for an earlier time, the job missed that run.
4. **The same key for the same trouble.** Every problem has a key such as `failed:backup`. DIFF compares the keys with the open alerts of the last run, kept in `alerts.json`: a key that is new is a new problem, a key that is gone is over, and a key that stays keeps the time it was first seen. That is what keeps the check quiet between the two messages.
5. **The mail.** MESSAGE builds one mail with the new problems, the ones that are over and the number still open. With `send_alerts` and a password from the Credential Manager, SEND delivers it at once through RETRY: three tries, with a pause of five seconds that doubles. In every other case, and when sending fails, it goes into the outbox.
6. **The numbers.** METRICS_TEXT\$ sets one gauge per job and value with the METRICS library and writes them in the text format of Prometheus to `metrics.prom` in the logs folder.

Run it

```
jdbasic health_check.jdb
4 jobs, 1 problem
  backup failed at 2026-10-09 16:31: exit 1: the drive E: is gone
In the outbox: alert 2026-10-09 164501.eml
```

A window lists the new problem. Fifteen minutes later the problem is still there, and the check prints it but stays silent. After the next backup works, one more alert follows: *AutomateWork: all clear*, with the backup under *Over now*.

The numbers file starts like this:

```
# HELP automatework_job_ok 1 when the last run worked.
# TYPE automatework_job_ok gauge
automatework_job_ok{job="downloads"} 1.0
automatework_job_ok{job="backup"} 0.0
```


Schedule it

The wizard plans the check every 15 minutes in the Windows Task Scheduler, on its own. Do not hand it to the job server: a check that the server starts cannot report that the server is gone.

Make it yours

The settings are in the part of `work.conf` that starts with `[health_checks]`:

```
[health_checks]
alert_to = ""
late_minutes = 30
server_minutes = 10
outbox_hours = 24
send_alerts = false
notify = "window"
```

An empty `alert_to` sends the alerts to your own address from the `[user]` part.

- **Long jobs.** The server marks its lock between jobs. A backup that runs for forty minutes keeps it from marking, so raise `server_minutes` above your longest job, for example to 60.
- **No window.** On a computer where nobody sits, set `notify = "none"`; the mail is enough.
- **A file that must be fresh.** Lena's Monday report must be younger than a week. One more check in `CHECK`, before `RETURN` out:

```
DIM report$ = "C:/Reports/monday.xlsx"
DIM info = FILE.STAT(report$)
DIM age = DATEDIFF("d", CDATE(info{"mtime"}), NOW())
IF age > 7 THEN
    Add(out, "old:monday", "old", "The Monday report is old.")
ENDIF
```

- **Your monitoring system.** If your IT department runs Prometheus with the Windows exporter, its text file collector reads `metrics.prom` from the folder it is told to watch. Point `[job_server]` folder there, or copy the file in a job.

When it goes wrong

- **"The job server is not running" right after logon:** the server starts with the logon and needs a moment. The next run of the check closes the alert again. To avoid the pair of mails, raise `server_minutes`.
- **The same alert again and again:** the problem comes and goes, for example a job that fails on every second run. Each failure after a good run is new. The log of the job server shows the pattern; fix the job.
- **"No mail password in the Credential Manager":** `send_alerts` is on, but X14 has not stored the password. Run `jdbasic secrets_audit.jdb --set mail`, or switch `send_alerts` off.
- **"Not sent: curl 7":** the mail server did not answer. The alert is in the outbox and goes out with the next `--send`.

Balance dividend

About 20 minutes a week of looking at logs, and no more Mondays that start with a phone call about a report that never came.

X14 Secrets and Audit Trail

The chore

When Lena took over the team's scripts, the mail password of the department was written into three of them, in plain text, and one of those files sat in a folder the whole floor could read. Changing the password meant finding every copy. A month later the auditors asked a different question: can she show that the backup really ran every Friday this quarter, and that nobody edited the log afterwards? She could show a log, but not that it was untouched.

What you get

Three tools in one program:

- **A safe place for secrets.** Passwords and keys go into the Windows Credential Manager, where Windows keeps them encrypted for your login. A recipe reads them from there; no file of yours holds them.
- **A search for written down secrets.** One command reads every script and settings file under your work folder and lists each line that sets a password, a token or a key, with the value masked.
- **An audit trail that shows changes.** The job server (X01) writes one signed line per run. One command checks every signature and the chain between the lines, and sums up what the jobs did.

```
jdbasic secrets_audit.jdb --verify
The audit trail is whole: 212 entries.
  backup: 13 runs, 1 failed, last 2026-10-09 16:31:07
  downloads: 7 runs, last 2026-10-09 18:30:02
```

Before you start

Until now no recipe wrote the mail password anywhere, and you typed it for every `--send`. That stays the default. This recipe is for the moment when a program has to work without you, such as the health check sending an alert at night. Then the password has to live somewhere, and the Credential Manager is the place Windows itself offers for that.

Check with your IT department whether stored passwords are allowed on your work computer. Some companies forbid it; then keep typing.

The program

The main program is a set of commands, one block each:

```
' Secrets and Audit Trail: keeps the passwords of your automations in
' the Windows Credential Manager instead of in files, finds passwords
' that are still written into scripts, and checks the signed audit
' trail the job server (X01) keeps of every run. The settings are the
' [secrets_audit] part of work.conf.
'
' jdbasic secrets_audit.jdb                which secrets are stored
' jdbasic secrets_audit.jdb --set mail      store a secret (asks)
' jdbasic secrets_audit.jdb --delete mail    remove a secret
' jdbasic secrets_audit.jdb --init-audit     make the signing key
' jdbasic secrets_audit.jdb --verify        check the audit trail
' jdbasic secrets_audit.jdb --scan          passwords in your files
```

```
IMPORT WORKCONF, JOBSTATE, VAULT, AUDIT, TRAIL
```

```
DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "secrets_audit."
DIM extra = WORKCONF.VALUE(cfg, key$ + "names", [])
DIM days = WORKCONF.VALUE(cfg, key$ + "summary_days", 7)
DIM home$ = "~/Documents/AutomateWork"
DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
DIM audit$ = PATH.JOIN$(JOBSTATE.FOLDER$(cfg), "audit.log")
DIM audit_target$ = TRAIL.TARGET$("audit-key")
DIM dry_run = WORKCONF.FLAG("dry-run")
```

```
DIM to_set$ = WORKCONF.ARG$("set")
IF to_set$ <> "" THEN
    DIM user$ = WORKCONF.ARG$("user")
    IF user$ = "" ANDALSO to_set$ = "mail" THEN
        user$ = WORKCONF.VALUE(cfg, "mail.user", "")
    ENDIF
    DIM first$ = VAULT.ASK$("Secret for " + to_set$ + ": ")
    DIM again$ = VAULT.ASK$("The same once more: ")
    IF first$ = "" ORELSE first$ <> again$ THEN
        PRINT "The two did not match; nothing was stored."
    ELSEIF dry_run THEN
        PRINT "would store "; TRAIL.TARGET$(to_set$)
    ELSE
        VAULT.PUT(TRAIL.TARGET$(to_set$), user$, first$)
        PRINT "Stored as "; TRAIL.TARGET$(to_set$)
```

```

    ENDIF
    END
ENDIF

DIM to_delete$ = WORKCONF.ARG$("delete")
IF to_delete$ <> "" THEN
    IF dry_run THEN
        PRINT "would remove "; TRAIL.TARGET$(to_delete$)
    ELSEIF VAULT.DELETE(TRAIL.TARGET$(to_delete$)) THEN
        PRINT "Removed "; TRAIL.TARGET$(to_delete$)
    ELSE
        PRINT "There is no secret "; TRAIL.TARGET$(to_delete$)
    ENDIF
    END
ENDIF

IF WORKCONF.FLAG("init-audit") THEN
    IF VAULT.HAS(audit_target$) THEN
        PRINT "The signing key exists already."
    ELSEIF dry_run THEN
        PRINT "would make a signing key in "; audit_target$
    ELSE
        VAULT.PUT(audit_target$, "audit", TRAIL.NEW_KEY$())
        PRINT "Made a signing key; the job server signs from now on."
    ENDIF
    END
ENDIF

IF WORKCONF.FLAG("verify") THEN
    DIM signing$ = VAULT.GET$(audit_target$)
    IF signing$ = "" THEN
        PRINT "No signing key yet. Run with --init-audit first."
        END
    ENDIF
    DIM check = AUDIT.VERIFY(audit$, signing$)
    DIM since_day = DATEADD("d", -days, NOW())
    DIM from_day$ = FORMAT_DATE(since_day, "%Y-%m-%d")
    DIM summary = TRAIL.SUMMARY(AUDIT.ENTRIES(audit$), from_day$)
    DIM lines = TRAIL.REPORT_LINES(check, summary)
    DIM i = 0
    FOR i = 0 TO LEN(lines) - 1
        PRINT lines[i]
    NEXT i
    IF NOT check{"ok"} THEN
        THROW "The audit trail does not check out."
    ENDIF
    END
ENDIF

```

```

IF WORKCONF.FLAG("scan") THEN
    DIM where$ = WORKCONF.ARG$("folder")
    IF where$ = "" THEN where$ = work$
    DIM hits = TRAIL.SCAN(WORKCONF.EXPAND$(where$))
    DIM found$ = STR$(LEN(hits)) + " lines look"
    IF LEN(hits) = 1 THEN found$ = "1 line looks"
    PRINT found$; " like a written down secret"
    DIM h = 0
    FOR h = 0 TO LEN(hits) - 1
        DIM hit = hits[h]
        PRINT " "; hit{"file"}; " line "; hit{"line"}
        PRINT " "; hit{"text"}
    NEXT h
END
ENDIF

DIM names = TRAIL.KNOWN(extra)
DIM n = 0
FOR n = 0 TO LEN(names) - 1
    DIM target$ = TRAIL.TARGET$(names[n])
    IF VAULT.HAS(target$) THEN
        PRINT " stored "; target$; " ("; VAULT.USER$(target$); ")"
    ELSE
        PRINT " missing "; target$
    ENDIF
NEXT n

```

The module names the secrets, searches for written down ones and sums up the trail:

```

' trail.jdb: the work of Secrets and Audit Trail. It names the secrets
' the recipes use, finds passwords written into scripts and settings,
' and sums up what the audit trail says the jobs did.
'
'   IMPORT TRAIL
'   DIM found = TRAIL.SCAN("C:/Users/lena/Documents/AutomateWork")
'   DIM week = TRAIL.SUMMARY(AUDIT.ENTRIES(path$), "2026-09-25")
'
' Every secret lives in the Credential Manager under "AutomateWork/"
' and its name, such as AutomateWork/mail.

EXPORT MODULE TRAIL

IMPORT SECRET

CONST PREFIX$ = "AutomateWork/"

```

```

' The Credential Manager name of a secret.
EXPORT FUNC TARGET$(name$)
    RETURN PREFIX$ + name$
ENDFUNC

' The secrets of the book's recipes and the extra names given.
EXPORT FUNC KNOWN(extra)
    DIM out = ["mail", "audit-key"]
    DIM k = 0
    FOR k = 0 TO LEN(extra) - 1
        DIM name$ = extra[k]
        IF NOT (name$ IN out) THEN PUSH(out, name$)
    NEXT k
    RETURN out
ENDFUNC

' A new random signing key.
EXPORT FUNC NEW_KEY$()
    RETURN SECRET.TOKEN$(32)
ENDFUNC

' The line with every quoted value replaced by stars.
EXPORT FUNC MASK$(line$)
    DIM q$ = CHR$(34)
    DIM quoted$ = q$ + "[" + q$ + "]" + q$
    RETURN REGEX.REPLACE(quoted$, line$, q$ + "****" + q$)
ENDFUNC

' TRUE when a line sets a password, secret, token or key to a value.
EXPORT FUNC LEAKY(line$)
    DIM q$ = CHR$(34)
    DIM names$ = "(?:password|passwd|pwd|pass|secret|token|api_?key)"
    DIM value$ = q$ + "[" + q$ + "]" + q$
    DIM pat$ = names$ + "[a-z_$]*\s*[:]\s*" + value$
    DIM hits = REGEX.FINDALL(pat$, LCASE$(line$))
    RETURN LEN(hits) > 0
ENDFUNC

' The lines of a text that look like a written down secret, as maps
' with "file", "line" and the masked "text".
EXPORT FUNC LEAKS(text$, file$)
    DIM out = []
    DIM lines = SPLIT(REPLACE$(text$, CHR$(13), ""), CHR$(10))
    DIM k = 0
    FOR k = 0 TO LEN(lines) - 1
        DIM line$ = lines[k]
        IF LEAKY(line$) THEN

```

```

        DIM hit AS MAP
        hit{"file"} = file$
        hit{"line"} = k + 1
        hit{"text"} = TRIM$(MASK$(line$))
        PUSH(out, hit)
    ENDIF
NEXT k
RETURN out
ENDFUNC

' TRUE for the kinds of files where people write passwords down.
FUNC Readable(name$)
    DIM ext$ = LCASE$(PATH.EXT$(name$))
    DIM kinds = [".jdb", ".toml", ".conf", ".ini", ".json", ".txt"]
    kinds = APPEND(kinds, [".bat", ".cmd", ".ps1"])
    RETURN ext$ IN kinds
ENDFUNC

FUNC Walk(folder$, out)
    DIM names = DIR$(PATH.JOIN$(folder$, "*"))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM name$ = names[k]
        DIM full$ = PATH.JOIN$(folder$, name$)
        DIM info = FILE.STAT(full$)
        IF NOT info{"hidden"} THEN
            IF info{"is_dir"} THEN
                Walk(full$, out)
            ELSEIF Readable(name$) AND ALSO info{"size"} < 1000000 THEN
                DIM hits = LEAKS(TXTREADER$(full$), full$)
                DIM h = 0
                FOR h = 0 TO LEN(hits) - 1
                    PUSH(out, hits[h])
                NEXT h
            ENDIF
        ENDIF
    NEXT k
    RETURN 0
ENDFUNC

' The written down secrets in every script and settings file under
' folder$.
EXPORT FUNC SCAN(folder$)
    DIM out = []
    IF FILE.ISDIR(folder$) THEN Walk(folder$, out)
    RETURN out
ENDFUNC

```



```

' Per job, the runs and failures of the audit entries from from$ on (a
' date or time text): a map of job name to {"runs", "fails", "last"}.
EXPORT FUNC SUMMARY(entries, from$)
    DIM out AS MAP
    DIM k = 0
    FOR k = 0 TO LEN(entries) - 1
        DIM e = entries[k]
        DIM at$ = e{"at"}
        IF at$ >= from$ AND ALSO MAP.EXISTS(e, "job") THEN
            DIM name$ = e{"job"}
            DIM row AS MAP
            row{"runs"} = 0
            row{"fails"} = 0
            IF MAP.EXISTS(out, name$) THEN row = out{name$}
            row{"runs"} = row{"runs"} + 1
            IF NOT e{"ok"} THEN row{"fails"} = row{"fails"} + 1
            row{"last"} = at$
            out{name$} = row
        ENDIF
    NEXT k
    RETURN out
ENDFUNC

' The verdict of AUDIT.VERIFY and a summary as lines for the console.
EXPORT FUNC REPORT_LINES(check, summary)
    DIM out = []
    IF check{"ok"} THEN
        DIM n$ = STR$(check{"entries"})
        PUSH(out, "The audit trail is whole: " + n$ + " entries.")
    ELSE
        PUSH(out, "The audit trail was changed:")
        DIM problems = check{"problems"}
        out = APPEND(out, SELECT(LAMBDA s$ -> " " + s$, problems))
    ENDIF
    DIM names = SORT(MAP.KEYS(summary))
    DIM j = 0
    FOR j = 0 TO LEN(names) - 1
        DIM name$ = names[j]
        DIM row = summary{name$}
        DIM line$ = " " + name$ + ": " + STR$(row{"runs"}) + " runs"
        DIM f$ = STR$(row{"fails"})
        IF row{"fails"} > 0 THEN line$ = line$ + ", " + f$ + " failed"
        PUSH(out, line$ + ", last " + row{"last"})
    NEXT j
    RETURN out
ENDFUNC

```

Two shared modules in `recipes/lib` do the rest. `vault.jdb` talks to the Credential Manager. `jdBasic` has no command for it, so the module declares the Windows functions it needs and calls them directly:

```
' vault.jdb: secrets in the Windows Credential Manager, where the
' system keeps them encrypted for the user who is logged in. A recipe
' stores a password once and reads it on every run; it never appears
' in a file of the recipe. The entries show up in the Control Panel
' under Credential Manager, Windows Credentials.
'
'   IMPORT VAULT
'   VAULT.PUT("AutomateWork/mail", "lena", password$)
'   DIM pw$ = VAULT.GET$("AutomateWork/mail")      ' "" when missing
'   PRINT VAULT.USER$("AutomateWork/mail")
'   VAULT.DELETE("AutomateWork/mail")
'   DIM env$ = "AUTOMATEWORK_MAIL"
'   DIM pw2$ = VAULT.FIND$("AutomateWork/mail", env$, "Password: ")

EXPORT MODULE VAULT

DECLARE FUNC LocalAlloc LIB "kernel32.dll" ALIAS "LocalAlloc" _
    (flags AS INTEGER, n AS INTEGER) AS INTEGER
DECLARE FUNC LocalFree LIB "kernel32.dll" ALIAS "LocalFree" _
    (p AS INTEGER) AS INTEGER
DECLARE FUNC GetLastError LIB "kernel32.dll" _
    ALIAS "GetLastError" () AS INTEGER
DECLARE SUB PutBytes LIB "kernel32.dll" ALIAS "RtlMoveMemory" _
    (dest AS INTEGER, src AS STRING, n AS INTEGER)
DECLARE FUNC GetByte LIB "kernel32.dll" ALIAS "RtlMoveMemory" _
    (dest AS RETURN, src AS INTEGER, n AS INTEGER) AS ARRAY
DECLARE FUNC CredWriteW LIB "advapi32.dll" ALIAS "CredWriteW" _
    (cred AS INTEGER, flags AS INTEGER) AS INTEGER
DECLARE FUNC CredReadW LIB "advapi32.dll" ALIAS "CredReadW" _
    (target AS STRING, kind AS INTEGER, flags AS INTEGER, _
    out_ptr AS INTEGER) AS INTEGER
DECLARE FUNC CredDeleteW LIB "advapi32.dll" ALIAS "CredDeleteW" _
    (target AS STRING, kind AS INTEGER, flags AS INTEGER) AS INTEGER
DECLARE SUB CredFree LIB "advapi32.dll" ALIAS "CredFree" _
    (p AS INTEGER)
```

Windows expects the entry as a block of 80 bytes with pointers to the name, the user and the secret. `PUT` builds that block byte by byte:

```
' Stores secret$ under target$ for the current Windows user, as UTF-16
' like the Control Panel does, replacing an entry of that name.
EXPORT SUB PUT(target$, user$, secret$)
```

```

IF target$ = "" THEN THROW "VAULT: a target name is needed"
DIM wide$ = Wide$(secret$)
DIM blob$ = LEFT$(wide$, LEN(wide$) - 2)
IF LEN(blob$) > MAX_BLOB THEN
    THROW "VAULT: a secret may have at most 1280 characters"
ENDIF
DIM p_target = Block(Wide$(target$))
DIM p_user = Block(Wide$(user$))
DIM p_blob = Block(blob$)
' The fields of CREDENTIALW, 80 bytes on 64-bit Windows.
DIM cred$ = Le$(0, 4) + Le$(CRED_TYPE_GENERIC, 4)
cred$ = cred$ + Le$(p_target, 8) + Le$(0, 8) + Le$(0, 8)
cred$ = cred$ + Le$(LEN(blob$), 4) + Le$(0, 4) + Le$(p_blob, 8)
cred$ = cred$ + Le$(CRED_PERSIST_LOCAL_MACHINE, 4) + Le$(0, 4)
cred$ = cred$ + Le$(0, 8) + Le$(0, 8) + Le$(p_user, 8)
DIM p_cred = Block(cred$)
DIM ok = CredWriteW(p_cred, 0)
DIM err = GetLastError()
DIM f = LocalFree(p_cred) + LocalFree(p_blob)
f = f + LocalFree(p_user) + LocalFree(p_target)
IF ok = 0 THEN THROW Failed$("store", target$, err)
ENDSUB

```

FIND\$ is what a recipe calls when it needs a secret. It looks in the Credential Manager first, then in an environment variable, and asks only when both are empty:

```

' A secret from the first place that has it: the Credential Manager
' under target$, the environment variable env$, or a prompt on the
' console. An empty env$ or prompt$ leaves that place out; "" when
' none of them answers.
EXPORT FUNC FIND$(target$, env$, prompt$)
    DIM stored$ = GET$(target$)
    IF stored$ <> "" THEN RETURN stored$
    IF env$ <> "" THEN
        DIM from_env$ = GETENV$(env$)
        IF from_env$ <> "" THEN RETURN from_env$
    ENDIF
    IF prompt$ = "" THEN RETURN ""
    RETURN ASK$(prompt$)

```

audit.jdb keeps the trail:

```

' audit.jdb: an audit trail that shows any change made to it. Every
' entry is one line, a JSON Web Token signed with a key only the owner
' has, and it carries the SHA-256 of the line before it. A changed
' line fails its signature; a removed or reordered line breaks the
' chain.
,
'   IMPORT AUDIT
'   AUDIT.APPEND("audit.log", key$, {"job": "backup", "ok": TRUE})
'   DIM check = AUDIT.VERIFY("audit.log", key$)
'   IF NOT check{"ok"} THEN PRINT JOIN(check{"problems"}, CHR$(10))
,
' Each entry holds the fields given plus seq (1, 2, 3 and so on), at
' (local time as text) and prev (the hash of the line before, "" for
' the first line).

EXPORT MODULE AUDIT

IMPORT JWT

' The lines of the trail, without empty ones.
EXPORT FUNC LINES(path$)
    IF NOT FILE.EXISTS(path$) THEN RETURN []
    DIM text$ = REPLACE$(TXTREADER$(path$), CHR$(13), "")
    DIM raw = SPLIT(text$, CHR$(10))
    DIM trimmed = SELECT(LAMBDA s$ -> TRIM$(s$), raw)
    RETURN FILTER(LAMBDA s$ -> s$ <> "", trimmed)
ENDFUNC

' Adds an entry with the fields of the map; answers its seq.
EXPORT FUNC APPEND(path$, key$, fields)
    IF key$ = "" THEN THROW "AUDIT: no signing key"
    DIM lines = LINES(path$)
    DIM claims AS MAP
    DIM names = MAP.KEYS(fields)
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        claims{names[k]} = fields{names[k]}
    NEXT k
    DIM seq = 1
    DIM prev$ = ""
    IF LEN(lines) > 0 THEN
        DIM last$ = lines[LEN(lines) - 1]
        DIM before = JWT.DECODE(last$)
        seq = before{"claims"}{"seq"} + 1
        prev$ = CODEC.SHA256$(last$)
    ENDIF
    claims{"seq"} = seq
    claims{"at"} = FORMAT_DATE(NOW(), "%Y-%m-%d %H:%M:%S")

```

```

    claims{"prev"} = prev$
    MKDIR PATH.DIRNAME$(path$)
    TXTWRITER path$, JWT.SIGN$(claims, key$) + CHR$(10), TRUE
    RETURN seq
ENDFUNC

' The entries as maps, read without checking them.
EXPORT FUNC ENTRIES(path$)
    DIM lines = LINES(path$)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO LEN(lines) - 1
        DIM d = JWT.DECODE(lines[k])
        PUSH(out, d{"claims"})
    NEXT k
    RETURN out
ENDFUNC

' Checks every signature and the chain. Answers {"ok", "entries",
' "problems"}, one problem per line that does not fit.
EXPORT FUNC VERIFY(path$, key$)
    DIM lines = LINES(path$)
    DIM problems = []
    DIM prev$ = ""
    DIM k = 0
    FOR k = 0 TO LEN(lines) - 1
        DIM where$ = "line " + STR$(k + 1)
        DIM check = JWT.VERIFY(lines[k], key$)
        IF NOT check{"ok"} THEN
            DIM why$ = " (" + check{"error"} + ")"
            DIM bad$ = ": the signature does not match"
            PUSH(problems, where$ + bad$ + why$)
        ELSE
            DIM claims = check{"claims"}
            IF claims{"seq"} <> k + 1 THEN
                DIM got$ = ": entry " + STR$(claims{"seq"})
                DIM want$ = " where " + STR$(k + 1) + " belongs"
                PUSH(problems, where$ + got$ + want$)
            ENDIF
            IF claims{"prev"} <> prev$ THEN
                DIM gap$ = ": the line before was changed or removed"
                PUSH(problems, where$ + gap$)
            ENDIF
        ENDIF
        prev$ = CODEC.SHA256$(lines[k])
    NEXT k
    DIM out AS MAP
    out{"ok"} = LEN(problems) = 0

```

```

    out{"entries"} = LEN(lines)
    out{"problems"} = problems
    RETURN out
ENDFUNC

```

How it works

1. **Names.** Every secret of the book has a name that starts with `AutomateWork/`: `AutomateWork/mail` for the mail password, `AutomateWork/audit-key` for the key that signs the trail. You find them in the Control Panel under *Credential Manager, Windows Credentials*, as generic credentials, and you can change or delete them there too.
2. **Storing.** `--set mail` asks twice, without showing what you type, and stores the secret only when both answers match. `VAULT.PUT` converts name and secret to UTF-16, the form Windows uses, copies them into memory Windows can read, fills in the 80 bytes of the entry and calls `CredWriteW`. Windows encrypts the secret with a key that belongs to your login.
3. **Reading.** `VAULT.GET$` calls `CredReadW`, which answers with a pointer to an entry. The module reads the size and the address of the secret out of it, copies the bytes and gives the memory back to Windows with `CredFree`.
4. **The search.** `TRAIL.SCAN` walks through the folder and its sub folders and reads every file that ends in `.jdb`, `.toml`, `.conf`, `.ini`, `.json`, `.txt`, `.bat`, `.cmd` or `.ps1`. A line counts when a word like *password*, *pwd*, *secret*, *token* or *api_key* is followed by `=` or `:` and a value in quotes. The report shows the line with every quoted value replaced by stars.
5. **Signing.** `--init-audit` makes a random key of 32 bytes and stores it as `AutomateWork/audit-key`. From then on the job server finds it and calls `AUDIT.APPEND` after each run. Every line is a JSON Web Token signed with that key, and it carries a number and the SHA-256 of the line before it.
6. **Checking.** `AUDIT.VERIFY` checks each signature with the key. A changed line fails its signature. A removed line leaves a gap in the numbers and a hash that points to the wrong line before it. `TRAIL.SUMMARY` then counts runs and failures per job for the last `summary_days` days.

Be clear about what this protects. The Credential Manager keeps your secrets out of files, backups and shared folders, and away from other people who log on to the computer. A program that runs under your own login can read them,

as the recipes do. The trail shows any change made by someone without the key. The key is in your Credential Manager, so the trail cannot prove anything against you yourself, and lines cut off at the end leave no gap. Compare the number of entries with what you expect, for example with the runs in the state file of the job server.

Run it

Without a switch the program lists the secrets the recipes know and whether each one is stored:

```
jdbasic secrets_audit.jdb
stored    AutomateWork/mail (lena)
missing   AutomateWork/audit-key
```

Storing the mail password, then the signing key:

```
jdbasic secrets_audit.jdb --set mail
Secret for mail: *****
The same once more: *****
Stored as AutomateWork/mail
jdbasic secrets_audit.jdb --init-audit
Made a signing key; the job server signs from now on.
```

The search over the work folder:

```
jdbasic secrets_audit.jdb --scan
1 line looks like a written down secret
C:/Users/lena/Documents/AutomateWork/config\old.ini line 4
password = "*****"
```

--folder searches another folder, for example the share where the team keeps its scripts.

Schedule it

The program needs no schedule; you run it when you set up a secret and when you want to see the trail. To have the trail checked every week, add the check to the jobs of the job server. A trail that does not check out ends the program with an error, so the job counts as failed and the health check (X13) tells you:

```
[[job]]
name = "audit check"
when = "weekly mon 07:00"
program = "X14_secrets_audit/secrets_audit.jdb"
args = ["--verify"]
```

Make it yours

The settings are in the part of `work.conf` that starts with `[secrets_audit]`:

```
[secrets_audit]
summary_days = 7
names = ["vpn", "ticket-token"]
```

- **Secrets of your own.** Every name in `names` shows up in the list and can be stored with `--set`. In your own recipe, read it with one line:

```
DIM token$ = VAULT.FIND$("AutomateWork/ticket-token", "", "Token: ")
```

- **An environment variable instead.** `FIND$` takes the variable you name when the Credential Manager has nothing. The health check reads `AUTOMATEWORK_MAIL`, which helps on a computer where the job runs under an account that has no stored credentials.
- **More words for the search.** The list of words is the text `names$` in `LEAKY`. To find connection strings as well, add `| connstr` before its closing bracket.

When it goes wrong

- **"The two did not match; nothing was stored":** type it again. The prompt is made for the keys of an English keyboard. For a password with umlauts or other letters, add it in the Control Panel instead: *Credential Manager, Windows Credentials, Add a generic credential*, with the name `AutomateWork/mail`.
- **"VAULT: Windows did not store":** Windows refused the entry, for example in a task that runs while nobody is logged on. Run `--set` in your own session at the keyboard. The error number in the message tells IT what Windows objected to.

- **"The audit trail was changed" with no one to blame:** the file was opened in an editor and saved. Some editors add marks at the start of a file or change characters, and every changed line fails its signature. Restore `audit.log` from the backup, and from then on only read it.
- **The search finds a line that is fine:** an example in a comment, for instance. Change the example to an empty value or to a call of `VAULT.FIND$`, and the line is no longer reported.

Balance dividend

About 10 minutes a week, and the hour it takes to find and change a password that is written into scripts, every time it changes.

X15 Balance Score

The chore

This book began with a question: where does your week go? Lena has an answer for every single part of it. The Time Tracker knows her hours, the calendar knows her meetings, the outbox knows when she wrote her mails. What she does not have is the picture of all three together, and how it changes. A week with 48 hours does not feel like one while she is in it; she notices in the third such week, when it is late.

What you get

A page on your own computer with one number for the week, from 0 to 100, and where the points went:

- **hours** above a normal week,
- **meetings** above a share of the week that is still fine,
- **work outside your working hours**, in the evening, early in the morning and at the weekend,
- **mails written at those times.**

Below it is the trend of the last eight weeks, and at the bottom the rules that make the score, with the weights of your `work.conf` filled in. Nothing on the page is hidden: you can recount every point.

Balance score, week 2026-W40

92

of 100

42.5

hours worked

7

in meetings

0

hours outside 08:30 to 17:30

3

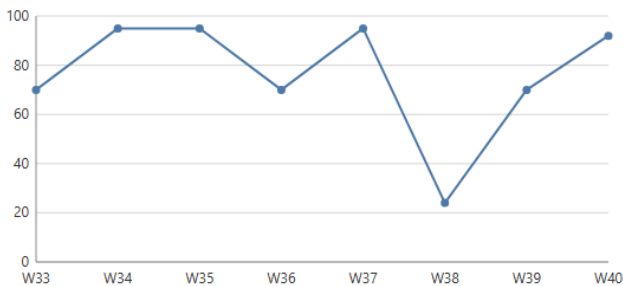
late mails

Where the points went

overtime -5

late mails -3

The last weeks



The balance score of a week, with the points lost and the trend

Before you start

The score is only as good as its sources:

- the **Time Tracker (E07)** has been running for a few weeks; the recipe finds its log through the folder of [time_tracker], as M15 does,
- your **calendar** as an .ics file. Outlook and Google export one; in Outlook choose *File, Save Calendar*, and save it as calendar.ics in your work folder,

- the **outbox** of the mail rule of Chapter 4, with its sent folder.

Your working hours are the `day_start` and `day_end` of the `[user]` part, the ones the wizard asked for. The recipe's own part holds the calendar and the weights:

```
[balance_score]
calendar = "~/Documents/AutomateWork/calendar.ics"
target_hours = 40
meeting_share = 40
weight_overtime = 2
weight_meetings = 1
weight_late = 3
weight_mail = 1
```

The program

The program reads the settings and either prints the scores, writes the page once, or serves it:

```
' Balance Score: your week as one number from 0 to 100, and the trend
' of the last weeks, on a web page on this computer. It reads the
' Time Tracker's log (E07), your calendar and the mails in the outbox.
' The settings are the [balance_score] part of work.conf, the working
' hours the [user] part.
'
'   jdbasic balance_score.jdb                serve until Ctrl+C
'   jdbasic balance_score.jdb --save my.html write the page once
'   jdbasic balance_score.jdb --print        the scores as text

IMPORT WORKCONF, BALANCE, OUTBOX, JDWEB, DT

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "balance_score."
DIM here$ = PATH.DIRNAME$(OS.ARGS()[0])
DIM time_home$ = "~/Documents/AutomateWork/time"
DIM time_dir$ = WORKCONF.FOLDER$(cfg, "time_tracker.folder", _
    time_home$)
DIM ics_home$ = "~/Documents/AutomateWork/calendar.ics"
DIM ics$ = WORKCONF.FOLDER$(cfg, key$ + "calendar", ics_home$)
DIM own$ = PATH.JOIN$(here$, "templates", "balance.html")
DIM tpl$ = WORKCONF.FOLDER$(cfg, key$ + "template", own$)

DIM rules AS MAP
rules{"offset"} = DT.LOCAL_OFFSET()
```

```

rules{"day_start"} = BALANCE.MINUTES(WORKCONF.VALUE(cfg, _
    "user.day_start", "08:30"))
rules{"day_end"} = BALANCE.MINUTES(WORKCONF.VALUE(cfg, _
    "user.day_end", "17:30"))
rules{"weeks"} = WORKCONF.VALUE(cfg, key$ + "weeks", 8)
rules{"target_hours"} = WORKCONF.VALUE(cfg, key$ + "target_hours", 40)
rules{"meeting_share"} = WORKCONF.VALUE(cfg, key$ + _
    "meeting_share", 40)
rules{"weight_overtime"} = WORKCONF.VALUE(cfg, key$ + _
    "weight_overtime", 2)
rules{"weight_meetings"} = WORKCONF.VALUE(cfg, key$ + _
    "weight_meetings", 1)
rules{"weight_late"} = WORKCONF.VALUE(cfg, key$ + "weight_late", 3)
rules{"weight_mail"} = WORKCONF.VALUE(cfg, key$ + "weight_mail", 1)

DIM log_file$ = PATH.JOIN$(time_dir$, "log.csv")
DIM sources = {"log": log_file$, "calendar": ics$}
sources{"outbox"} = OUTBOX.FOLDER$(cfg)

IF WORKCONF.FLAG("print") THEN
    DIM n = rules{"weeks"}
    DIM weeks = BALANCE.WEEKS(sources, rules, DT.NOW(), n)
    DIM k = 0
    FOR k = 0 TO LEN(weeks) - 1
        DIM w = weeks[k]
        DIM week$ = w{"week"}
        PRINT week$; " "; FORMAT$("{:>3}", w{"score"}); " "; _
            FORMAT$("{:.1f}", w{"hours"}); " hours"
    NEXT k
END
ENDIF

IF NOT FILE.EXISTS(tpl$) THEN THROW "No page template at " + tpl$
DIM save$ = WORKCONF.ARG$("save")
IF save$ <> "" THEN
    DIM all = BALANCE.WEEKS(sources, rules, DT.NOW(), rules{"weeks"})
    TXTWRITER save$, BALANCE.PAGE$(all, rules, tpl$)
    PRINT "wrote "; save$
END
ENDIF

DIM port = WORKCONF.VALUE(cfg, key$ + "port", 8766)
BALANCE.MOUNT(sources, rules, tpl$)
IF NOT JDWEB.LISTEN(port) THEN
    THROW "Port " + STR$(port) + " is taken. Change it in work.conf."
ENDIF
PRINT "Your balance score: http://localhost:"; port; "/"

```

```
PRINT "Only this computer can open it. Stop it with Ctrl+C."
HTTP.SERVER.WAIT
```

The module collects the numbers, makes the score and builds the page:

```
' balance.jdb: a weekly balance score from the hours in the Time
' Tracker's log, the meetings in a calendar and the mails you wrote.
' The score starts at 100 and loses points for overtime, for a week
' that is mostly meetings, for work outside your working hours and
' for mails written then.
,
'   IMPORT BALANCE
'   DIM weeks = BALANCE.WEEKS(sources, rules, now, 8)
'   PRINT BALANCE.SCORE(weeks[7], rules){ "score" }
'   BALANCE.MOUNT(sources, rules, "templates/balance.html")

EXPORT MODULE BALANCE

IMPORT DT, ICAL, DF, SVG, TMPL, JDWEB

DIM gSources AS MAP
DIM gRules AS MAP
DIM gTemplate$ = ""

' The ISO week of an instant as "2026-W41".
EXPORT FUNC WEEK$(t, offset)
    DIM w = DT.ISOWEEK(t, offset)
    RETURN STR$(w{"year"}) + "-W" + RIGHT$("0" + STR$(w{"week"}), 2)
ENDFUNC

' The minutes after midnight of a time of day such as "17:30".
EXPORT FUNC MINUTES(hh_mm$)
    DIM parts = SPLIT(hh_mm$, ":")
    RETURN VAL(parts[0]) * 60 + VAL(parts[1])
ENDFUNC

' TRUE for an instant on a weekend or outside the working hours of
' rules ("day_start" and "day_end" in minutes, "offset" in hours).
EXPORT FUNC OUTSIDE(t, rules)
    DIM p = DT.PARTS(t, rules{"offset"})
    IF p{"weekday"} = 0 ORELSE p{"weekday"} = 6 THEN RETURN TRUE
    DIM m = p{"hour"} * 60 + p{"minute"}
    RETURN m < rules{"day_start"} ORELSE m >= rules{"day_end"}
ENDFUNC

SUB Add(totals, key$, name$, v)
    DIM slot$ = key$ + "|" + name$
```

```

    DIM had = 0.0
    IF MAP.EXISTS(totals, slot$) THEN had = totals[slot$]
    totals[slot$] = had + v
ENDSUB

' The numbers of one week from the totals, as a map with "week",
' "hours", "late_hours", "meeting_hours", "mails" and "late_mails".
EXPORT FUNC WEEK_OF(totals, key$)
    DIM w AS MAP
    w{"week"} = key$
    DIM names = ["hours", "late_hours", "meeting_hours", "mails", _
        "late_mails"]
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM name$ = names[k]
        DIM slot$ = key$ + "|" + name$
        DIM v = 0.0
        IF MAP.EXISTS(totals, slot$) THEN v = totals[slot$]
        w{name$} = v
    NEXT k
    RETURN w
ENDFUNC

SUB AddWork(totals, t_from, t_to, rules)
    DIM quarter = 900
    DIM t = t_from
    DO WHILE t < t_to
        DIM key$ = WEEK$(t, rules{"offset"})
        Add(totals, key$, "hours", 0.25)
        IF OUTSIDE(t, rules) THEN
            Add(totals, key$, "late_hours", 0.25)
        ENDIF
        t = t + quarter
    LOOP
ENDSUB

' Adds the work of a time log (time,event,project) to totals, a map of
' numbers per week, counted in quarter hours. A block still open runs
' until now.
EXPORT SUB ADD_LOG(totals, log_path$, rules, now)
    IF NOT FILE.EXISTS(log_path$) THEN RETURN
    DIM kinds = ["STRING", "STRING", "STRING"]
    DIM rows = CSVREADER(log_path$, ",", TRUE, kinds)
    DIM offset = rules{"offset"}
    DIM open = -1.0
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM r = rows[k]

```

```

    DIM when$ = r[0]
    DIM what$ = r[1]
    DIM t = DT.PARSE(when$, offset)
    IF open >= 0 THEN AddWork(totals, open, t, rules)
    open = -1.0
    IF what$ = "start" THEN open = t
NEXT k
    IF open >= 0 THEN AddWork(totals, open, now, rules)
ENDSUB

' Adds the hours of the calendar's meetings between two instants; days
' blocked whole are left out.
EXPORT SUB ADD_MEETINGS(totals, ics_path$, rules, t_from, t_to)
    IF NOT FILE.EXISTS(ics_path$) THEN RETURN
    DIM cal = ICAL.PARSEFILE(ics_path$)
    DIM frame = ICAL.EXPAND(cal, t_from, t_to)
    DIM k = 0
    FOR k = 0 TO DF.NROWS(frame) - 1
        DIM r = DF.ROW(frame, k)
        IF NOT ICAL.ALLDAY(cal, r{"event"}) THEN
            DIM key$ = WEEK$(r{"start"}, rules{"offset"})
            DIM h = (r{"end"} - r{"start"}) / 3600
            Add(totals, key$, "meeting_hours", h)
        ENDIF
    NEXT k
ENDSUB

' Adds the mails of the .eml files in a folder, each counted at the
' time of its file.
EXPORT SUB ADD_MAILS(totals, folder$, rules)
    IF NOT FILE.ISDIR(folder$) THEN RETURN
    DIM names = DIR$(PATH.JOIN$(folder$, "*.eml"))
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM file$ = names[k]
        DIM info = FILE.STAT(PATH.JOIN$(folder$, file$))
        DIM t = DT.PARSE(info{"mtime"}, rules{"offset"})
        DIM key$ = WEEK$(t, rules{"offset"})
        Add(totals, key$, "mails", 1)
        IF OUTSIDE(t, rules) THEN Add(totals, key$, "late_mails", 1)
    NEXT k
ENDSUB

' The score of one week and its parts: a map with "score" (0 to 100)
' and "parts", one [reason, points lost] per rule that cost points.
EXPORT FUNC SCORE(w, rules)
    DIM parts = []
    DIM over = w{"hours"} - rules{"target_hours"}

```



```

IF over > 0 THEN
    PUSH(parts, ["overtime", over * rules{"weight_overtime"}])
ENDIF
IF w{"hours"} > 0 THEN
    DIM share = w{"meeting_hours"} / w{"hours"} * 100
    DIM above = share - rules{"meeting_share"}
    IF above > 0 THEN
        DIM busy = above * rules{"weight_meetings"}
        PUSH(parts, ["meetings", busy])
    ENDIF
ENDIF
IF w{"late_hours"} > 0 THEN
    DIM late = w{"late_hours"} * rules{"weight_late"}
    PUSH(parts, ["outside working hours", late])
ENDIF
IF w{"late_mails"} > 0 THEN
    DIM mails = w{"late_mails"} * rules{"weight_mail"}
    PUSH(parts, ["late mails", mails])
ENDIF
DIM lost = SUM(SELECT(LAMBDA p -> p[1], parts))
DIM score = ROUND(100 - lost)
IF score < 0 THEN score = 0
RETURN {"score": score, "parts": parts}
ENDFUNC

' The last n weeks up to the week of now, oldest first, each with the
' numbers of the sources ("log", "calendar", "outbox") and its score.
EXPORT FUNC WEEKS(sources, rules, now, n)
    DIM offset = rules{"offset"}
    DIM this_week = DT.STARTOF(now, "week", offset)
    DIM first = DT.ADD(this_week, "-" + STR$(n - 1) * 7) + " days", _
        offset)
    DIM totals AS MAP
    ADD_LOG(totals, sources{"log"}, rules, now)
    ADD_MEETINGS(totals, sources{"calendar"}, rules, first, now)
    DIM box$ = sources{"outbox"}
    ADD_MAILS(totals, box$, rules)
    ADD_MAILS(totals, PATH.JOIN$(box$, "sent"), rules)
    DIM out = []
    DIM k = 0
    FOR k = 0 TO n - 1
        DIM t = DT.ADD(first, STR$(k * 7) + " days", offset)
        DIM key$ = WEEK$(t, offset)
        DIM w = WEEK_OF(totals, key$)
        DIM s = SCORE(w, rules)
        w{"score"} = s{"score"}
        w{"parts"} = s{"parts"}
        w{"monday"} = DT.FMT$(t, "%Y-%m-%d", offset)
    
```

```

        PUSH(out, w)
    NEXT k
    RETURN out
ENDFUNC

' The scores of the weeks as a line chart, SVG markup.
EXPORT FUNC CHART$(weeks)
    DIM labels = SELECT(LAMBDA w -> MID$(w{"week"}, 5, 4), weeks)
    DIM values = SELECT(LAMBDA w -> w{"score"}, weeks)
    DIM c = SVG.CHART("line", 560, 260)
    SVG.SETOPT(c, "legend", "none")
    SVG.SETOPT(c, "values", 1)
    SVG.SETOPT(c, "y_min", 0)
    SVG.SETOPT(c, "y_max", 100)
    SVG.LABELS(c, labels)
    SVG.SERIES(c, "score", values)
    RETURN SVG.MARKUP$(SVG.RENDER(c))
ENDFUNC

' Minutes after midnight as a time of day such as "08:30".
EXPORT FUNC CLOCK$(minutes)
    DIM h$ = RIGHT$("0" + STR$(INT(minutes / 60)), 2)
    RETURN h$ + ":" + RIGHT$("0" + STR$(minutes MOD 60), 2)
ENDFUNC

FUNC One$(v)
    RETURN STR$(ROUND(v * 10) / 10)
ENDFUNC

' The page: this week's score and its parts, the trend and the rules.
EXPORT FUNC PAGE$(weeks, rules, template$)
    DIM now_w = weeks[LEN(weeks) - 1]
    DIM lost = SELECT(LAMBDA p -> _
        {"why": p[0], "points": One$(p[1])}, now_w{"parts"})
    DIM model = {"week": now_w{"week"}, "score": now_w{"score"}}
    model{"hours"} = One$(now_w{"hours"})
    model{"meeting_hours"} = One$(now_w{"meeting_hours"})
    model{"late_hours"} = One$(now_w{"late_hours"})
    model{"late_mails"} = now_w{"late_mails"}
    model{"lost"} = lost
    model{"chart"} = CHART$(weeks)
    model{"rules"} = rules
    model{"day_start"} = CLOCK$(rules{"day_start"})
    model{"day_end"} = CLOCK$(rules{"day_end"})
    RETURN TMPL.RENDER$(template$, model)
ENDFUNC

FUNC ShowPage(request)

```

```

    DIM weeks = WEEKS(gSources, gRules, DT.NOW(), gRules{"weeks"})
    RETURN PAGE$(weeks, gRules, gTemplate$)
ENDFUNC

FUNC ShowData(request)
    DIM n = gRules{"weeks"}
    RETURN {"weeks": WEEKS(gSources, gRules, DT.NOW(), n)}
ENDFUNC

' Registers the page at / and the numbers at /data.json.
EXPORT SUB MOUNT(sources, rules, template$)
    gSources = sources
    gRules = rules
    gTemplate$ = template$
    JDWEB.GET("/", ShowPage@)
    JDWEB.GET("/data.json", ShowData@)
ENDSUB

```

The page is a template in `templates/balance.html`, so you can change its look without touching the code. This is the part that explains the score; `TMPL.RENDER$` fills in the parts in double braces:

```

</div>
<div class="box">
    <h2>The last weeks</h2>
    {{{ chart }}}
</div>
<div class="box">
    <h2>How the score is made</h2>
    <p>Every week starts at 100 points. It loses</p>
    <ul>
    <li>{{ rules.weight_overtime }} for every hour worked above
        {{ rules.target_hours }},</li>
    <li>{{ rules.weight_meetings }} for every percent of the hours
        spent in meetings above {{ rules.meeting_share }} percent,</li>
    <li>{{ rules.weight_late }} for every hour worked before
        {{ day_start }}, after {{ day_end }} or on a weekend,</li>
    <li>{{ rules.weight_mail }} for every mail written at those

```

How it works

1. **Instants instead of dates.** The module counts in DT instants, seconds since 1970 in UTC, and turns them into a wall clock with the offset of your time zone. An hour is always 3600 seconds, and no change of the clocks shifts a date.

2. **Weeks.** WEEK\$ names the ISO week of an instant, such as 2026-W41. The week starts on Monday, the way most calendars in Europe count it.
3. **Working hours.** OUTSIDE says whether an instant falls on a weekend or before day_start or after day_end, both in minutes after midnight from MINUTES.
4. **The time log.** ADD_LOG reads the log of E07 and walks through every block of work in steps of a quarter of an hour. Each step adds a quarter to the hours of its week, and to the late hours if it is outside. A block that runs over midnight or over the end of a week is split on its own.
5. **Totals.** The numbers go into one map under keys such as 2026-W41|hours. WEEK_OF collects the numbers of one week from it, with 0 for anything that has not happened.
6. **Meetings and mails.** ADD_MEETINGS expands the calendar with ICAL.EXPAND, so a weekly meeting counts every week, and leaves out all-day events such as holidays. ADD_MAILS counts the .eml files of the outbox and of sent at the time of their file.
7. **The score.** SCORE starts at 100 and takes away the points of each rule that applies, and it keeps the reasons, so the page can show them. The score never goes below 0.
8. **The page.** PAGE\$ fills the template with this week's numbers, the reasons, the chart of CHART\$ and the rules. MOUNT puts the page at / and the numbers as JSON at /data.json, and every request counts again, so a refresh shows the minute you are in.

Run it

```
jdbasic balance_score.jdb --print
2026-W33    70  48.5 hours
2026-W34    95  42.5 hours
2026-W35    95  42.5 hours
2026-W36    70  48.5 hours
2026-W37    95  42.5 hours
2026-W38    24  57.5 hours
2026-W39    70  48.5 hours
2026-W40    92  42.5 hours

jdbasic balance_score.jdb
```

```
Your balance score: http://localhost:8766/
Only this computer can open it. Stop it with Ctrl+C.
```

Open the address in your browser. `--save week.html` writes the page once into a file, for a week you want to keep or show.

Schedule it

The wizard starts the page when you log on, like the dashboard of M08. It costs nothing while nobody looks at it. Leave it out of the X01 job server: a job server runs jobs that end, and this one serves until you stop it.

Make it yours

- **Your own weights.** The weights are settings. If evenings hurt you more than long weeks, raise `weight_late` and lower `weight_overtime`. Change them for a month before you trust the number; the trend shows whether the score agrees with how the weeks felt.
- **A part-time week.** Set `target_hours` to the hours of your contract, 24 for three days a week.
- **A rule of your own.** Each rule is four lines in SCORE. This one costs a point for every meeting hour above ten a week:

```
DIM much = w{"meeting_hours"} - 10
IF much > 0 THEN PUSH(parts, ["too many meetings", much])
```

Add a line to the template that explains it, so the page stays honest.

- **More weeks.** `weeks = 12` in `[balance_score]` shows a quarter of a year.

When it goes wrong

- **The score is 100 every week:** the recipe finds no log. Check the folder of `[time_tracker]`, and that `log.csv` in it has lines.
- **No meetings:** the calendar file is old or not where calendar says. Export it again; the recipe reads the file, not your mail program.
- **Hours shifted by one:** the recipe uses the offset of your time zone today for all weeks, so a log line from before a change of the clocks is read one hour

off. Up to an hour of such a week can move across the edge of your working hours. The weeks before the change look a little better or worse than they were.

- **"Port 8766 is taken":** another program uses the port. Set port in [balance_score] to another number above 1024.

Balance dividend

About 15 minutes a week, for the look at the numbers you would otherwise put together by hand. The real dividend is the week you notice in time.

Bonus: X16 Desktop Robot

The chore

Every Friday Lena moves the corrections of the week from the bookings report into an old cost centre program. The company that wrote it is long gone, and the program can neither import a file nor talk to other programs. Forty corrections, each a window to open, four fields to fill and a button to press. A macro of the kind M16 plays would save the typing, but Lena does not trust a macro that clicks blindly into a program that sometimes opens a different dialog.

What you get

The Desktop Robot works the program from a job file. It waits for the window, brings it to the front, and before it clicks it looks at the screen: a region of the window has to match a reference picture, so a click only happens when the right button is where it should be. Every step may be tried again, every step writes a line to the log, and a job that fails ends with an error the job server of X01 notices.

Before you start

The setup wizard has run. Copy `order_entry.toml` and `new_button.png` from the recipe's folder into `~/Documents/AutomateWork/robot`, the folder the job setting names, and change the job for your program. The points of a job count from the top-left corner of the window's inside, so a job still works when the window moves. Measure them with `jdbasic macro_player.jdb --where (M16)`: point at the top-left corner of the window's inside first, then at each place, and subtract the first position from the others.

For each check, take a reference picture of the region once, while the program shows what the robot should see. A few lines do it; the numbers are the region of the check:

```
IMPORT IMG
DIM title$ = "Order Entry - Version 4"
DIM rc = OS.SCREENSHOT("C:/temp/window.png", "client", title$)
DIM shot = IMG.READPNG("C:/temp/window.png")
DIM button = IMG.CROP(shot, 12, 44, 96, 24)
IMG.WRITEPNG(button, "new_button.png")
```

Watch out

While a job runs, the robot takes over your keyboard and your mouse. To stop it at once, hold Esc, or push the mouse into the top-left corner of the screen; it checks both before every single action. Windows delivers key strokes and clicks only to an unlocked screen with you logged in, so a job needs a computer that is on, unlocked and nobody else's to use. Do not leave a screen unlocked for a robot to run. Ask your IT department first: some companies forbid programs that type and click for you, and some will rather give you a computer of its own for this.

The program

The program reads the settings and the job, and either shows the steps or runs them with a log:

```
' Desktop Robot: works a desktop program that has no import and no
' interface for other programs, the way a person would, from a job
' file. The settings are the [desktop_robot] part of work.conf.
'
'   jdbasic desktop_robot.jdb           run the job after a countdown
'   jdbasic desktop_robot.jdb --dry-run show the steps of the job
'   jdbasic desktop_robot.jdb --job other.toml  another job file

IMPORT WORKCONF, LOGGER, INPUTKIT, ROBOT

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "desktop_robot."
DIM home$ = "~/Documents/AutomateWork"
DIM job_home$ = home$ + "/robot/order_entry.toml"
DIM job_path$ = WORKCONF.FOLDER$(cfg, key$ + "job", job_home$)
IF WORKCONF.ARG$("job") <> "" THEN
    job_path$ = WORKCONF.EXPAND$(WORKCONF.ARG$("job"))
ENDIF
DIM wait_s = WORKCONF.VALUE(cfg, key$ + "countdown", 5)
DIM dry_run = WORKCONF.FLAG("dry-run")

DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
DIM logs$ = PATH.JOIN$(work$, "logs")
MKDIR logs$
DIM lg = LOGGER.NEW("desktop_robot")
LOGGER.TO_CONSOLE(lg, FALSE)
LOGGER.TO_FILE(lg, PATH.JOIN$(logs$, "desktop_robot.log"))
```



```

DIM job = ROBOT.LOAD(job_path$)
DIM errors = job{"errors"}
IF LEN(errors) > 0 THEN
    PRINT "The job has mistakes:"
    PRINT " "; JOIN(errors, CHR$(10) + " ")
    END 1
ENDIF
DIM plan = ROBOT.PLAN_LINES(job)
IF dry_run THEN
    DIM title$ = job{"window"}
    PRINT "Job "; PATH.BASENAME$(job_path$); " for "; title$; ":"
    PRINT " "; JOIN(plan, CHR$(10) + " ")
    PRINT LEN(plan); " steps. Without --dry-run the robot runs them."
    END
ENDIF

LOGGER.INFO(lg, "job started", {"job": job_path$})
INPUTKIT.COUNTDOWN(wait_s)
DIM r = ROBOT.RUN(job, lg, logs$)
IF NOT r{"ok"} THEN
    DIM why = {"step": r{"step"}, "why": r{"why"}}
    LOGGER.ERROR(lg, "job stopped", why)
    END 1
ENDIF
LOGGER.INFO(lg, "job done", {"steps": LEN(plan)})

```

A job is a TOML file; the comment at its top lists every kind of step:

```

# A job of the Desktop Robot (X16): enters one order into the old
# order program. Points and regions count from the top-left corner of
# the window's inside; find them: jdbasic macro_player.jdb --where
#
#   do = "wait_window"    wait for the window, timeout_s (30)
#   do = "check"          region [x, y, width, height] must look like
#                           image (a PNG next to this file),
#                           tolerance (10)
#   do = "click"          at [x, y]; also "doubleclick"
#   do = "type"           text
#   do = "key"            keys, such as "tab" or "ctrl+s"
#   do = "pause"          ms
#   tries                 how often to try a step before the job
#                           stops (1)
#
window = "Order Entry - Version 4"

[[step]]
do = "wait_window"

```

```

timeout_s = 60

[[step]]
do = "check"
region = [12, 44, 96, 24]
image = "new_button.png"
tolerance = 12
tries = 5

[[step]]
do = "click"
at = [60, 56]

[[step]]
do = "pause"
ms = 800

[[step]]
do = "type"
text = "Müller GmbH"

[[step]]
do = "key"
keys = "tab"

[[step]]
do = "type"
text = "Copy paper, 10 packs"

[[step]]
do = "key"
keys = "ctrl+s"
tries = 2

```

The module reads the job, compares screen regions and runs the steps:

```

' robot.jdb: a desktop robot for a program without an interface for
' other programs. A job file names the window and the steps; points
' and screen regions count from the top-left corner of the window's
' inside, so the job still works when the window moves. Before a click
' the robot can compare a region of the window with a reference
' picture, and every step may be tried more than once.
'
'   IMPORT ROBOT
'   DIM job = ROBOT.LOAD("order_entry.toml")
'   PRINT JOIN(ROBOT.PLAN_LINES(job), CHR$(10))
'   DIM r = ROBOT.RUN(job, lg, "C:/temp/shots")

```

```

EXPORT MODULE ROBOT

IMPORT CONF, IMG, INPUTKIT, LOGGER

DIM gKinds$ = "|wait_window|check|click|doubleclick|type|key|pause|"

' A list of n numbers from a job field, or 0 when it is not one.
FUNC Numbers(v, n)
  IF TYPEOF(v) <> "ARRAY" THEN RETURN 0
  IF LEN(v) <> n THEN RETURN 0
  DIM k = 0
  FOR k = 0 TO n - 1
    DIM kind$ = TYPEOF(v[k])
    DIM tag$ = "|" + kind$ + "|"
    IF INSTR("|INT64|INTEGER|FLOAT64|", tag$) < 0 THEN
      RETURN 0
    ENDIF
  NEXT k
  RETURN v
ENDFUNC

' A field of a step, or the fallback when the step leaves it out.
FUNC Field(s, name$, fallback)
  IF MAP.EXISTS(s, name$) THEN RETURN s{name$}
  RETURN fallback
ENDFUNC

' The job in a TOML text as {"window", "steps", "errors"}; pictures
' are looked up in base_dir$. Each step gets "do", "line" (its number
' from 1), "tries" and the fields its kind needs.
EXPORT FUNC PARSE_JOB(toml$, base_dir$)
  DIM t = CONF.TOML_TEXT(toml$)
  DIM errors = []
  DIM steps = []
  DIM window$ = ""
  IF MAP.EXISTS(t, "window") THEN window$ = t{"window"}
  IF window$ = "" THEN PUSH(errors, "the job names no window")
  DIM raw = []
  IF MAP.EXISTS(t, "step") THEN raw = t{"step"}
  IF LEN(raw) = 0 THEN PUSH(errors, "the job has no [[step]]")
  DIM k = 0
  FOR k = 0 TO LEN(raw) - 1
    DIM r = raw[k]
    DIM where$ = "step " + STR$(k + 1) + ": "
    DIM do$ = LCASE$(Field(r, "do", ""))
    DIM s AS MAP
    s{"do"} = do$

```

```

s{"line"} = k + 1
s{"tries"} = Field(r, "tries", 1)
IF INSTR(gKinds$, "|" + do$ + "|") < 0 THEN
    PUSH(errors, where$ + "unknown do = '" + do$ + "'")
ELSEIF do$ = "wait_window" THEN
    s{"timeout_s"} = Field(r, "timeout_s", 30)
    PUSH(steps, s)
ELSEIF do$ = "check" THEN
    DIM region = Numbers(Field(r, "region", 0), 4)
    DIM pic$ = Field(r, "image", "")
    DIM path$ = PATH.JOIN$(base_dir$, pic$)
    IF TYPEOF(region) <> "ARRAY" THEN
        DIM need$ = "region needs [x, y, width, height]"
        PUSH(errors, where$ + need$)
    ELSEIF pic$ = "" ORELSE NOT FILE.EXISTS(path$) THEN
        PUSH(errors, where$ + "no picture " + path$)
    ELSE
        s{"region"} = region
        s{"image"} = path$
        s{"tolerance"} = Field(r, "tolerance", 10)
        PUSH(steps, s)
    ENDIF
ELSEIF do$ = "click" ORELSE do$ = "doubleclick" THEN
    DIM at = Numbers(Field(r, "at", 0), 2)
    IF TYPEOF(at) <> "ARRAY" THEN
        PUSH(errors, where$ + do$ + " needs at = [x, y]")
    ELSE
        s{"at"} = at
        PUSH(steps, s)
    ENDIF
ELSEIF do$ = "type" THEN
    DIM text$ = Field(r, "text", "")
    IF text$ = "" THEN
        PUSH(errors, where$ + "type needs a text")
    ELSE
        s{"text"} = text$
        PUSH(steps, s)
    ENDIF
ELSEIF do$ = "key" THEN
    DIM keys$ = Field(r, "keys", "")
    DIM c = INPUTKIT.COMBO(keys$, FALSE)
    IF c{"error"} <> "" THEN
        PUSH(errors, where$ + c{"error"})
    ELSE
        s{"keys"} = keys$
        PUSH(steps, s)
    ENDIF
ELSEIF do$ = "pause" THEN

```

```

        s{"ms"} = Field(r, "ms", 500)
        PUSH(steps, s)
    ENDIF
NEXT k
RETURN {"window": window$, "steps": steps, "errors": errors}
ENDFUNC

' The job in a file; pictures sit next to it.
EXPORT FUNC LOAD(path$)
    IF NOT FILE.EXISTS(path$) THEN THROW "ROBOT: no job file " + path$
    RETURN PARSE_JOB(TXTREADER$(path$), PATH.DIRNAME$(path$))
ENDFUNC

' One step as a line of a plan.
EXPORT FUNC PLAN_LINE$(s, window$)
    DIM do$ = s{"do"}
    DIM head$ = "step " + STR$(s{"line"}) + ": "
    DIM out$ = head$ + do$
    IF do$ = "wait_window" THEN
        out$ = head$ + "wait up to " + STR$(s{"timeout_s"})
        out$ = out$ + " s for '" + window$ + "'"
    ELSEIF do$ = "check" THEN
        DIM rg = s{"region"}
        out$ = head$ + "check " + STR$(rg[2]) + " x " + STR$(rg[3])
        out$ = out$ + " at " + STR$(rg[0]) + ", " + STR$(rg[1])
        out$ = out$ + " against " + PATH.BASENAME$(s{"image"})
    ELSEIF do$ = "click" OR ELSE do$ = "doubleclick" THEN
        DIM at = s{"at"}
        out$ = head$ + do$ + " at " + STR$(at[0]) + ", " + STR$(at[1])
    ELSEIF do$ = "type" THEN
        out$ = head$ + "type '" + s{"text"} + "'"
    ELSEIF do$ = "key" THEN
        out$ = head$ + "press " + s{"keys"}
    ELSEIF do$ = "pause" THEN
        out$ = head$ + "pause " + STR$(s{"ms"}) + " ms"
    ENDIF
    IF s{"tries"} > 1 THEN
        out$ = out$ + ", " + STR$(s{"tries"}) + " tries"
    ENDIF
    RETURN out$
ENDFUNC

' Every step of a job as a line of a plan.
EXPORT FUNC PLAN_LINES(job)
    DIM window$ = job{"window"}
    RETURN SELECT(LAMBDA USE(window$) s -> PLAN_LINE$(s, window$), _
        job{"steps"})
ENDFUNC

```

```

' A point inside the window as a point of the screen.
EXPORT FUNC SCREEN_POINT(origin, at)
    RETURN [origin[0] + at[0], origin[1] + at[1]]
ENDFUNC

' How far a region of a screenshot is from a reference picture: the
' mean difference of the red, green and blue values, 0 to 255. A
' region that does not fit, or a picture of another size, counts 255.
EXPORT FUNC REGION_DIFF(shot, ref, region)
    DIM rx = region[0]
    DIM ry = region[1]
    DIM w = region[2]
    DIM h = region[3]
    DIM sw = shot{"width"}
    IF ref{"width"} <> w ORELSE ref{"height"} <> h THEN RETURN 255
    IF rx + w > sw ORELSE ry + h > shot{"height"} THEN RETURN 255
    DIM cs = shot{"channels"}
    DIM cr = ref{"channels"}
    DIM ps = shot{"pixels"}
    DIM pr = ref{"pixels"}
    DIM total = 0.0
    DIM x = 0
    DIM y = 0
    DIM ch = 0
    FOR y = 0 TO h - 1
        FOR x = 0 TO w - 1
            DIM at_s = ((ry + y) * sw + rx + x) * cs
            DIM at_r = (y * w + x) * cr
            FOR ch = 0 TO 2
                DIM s_ch = ch
                IF cs < 3 THEN s_ch = 0
                DIM r_ch = ch
                IF cr < 3 THEN r_ch = 0
                DIM vs = ps[at_s + s_ch]
                DIM vr = pr[at_r + r_ch]
                total = total + ABS(vs - vr)
            NEXT ch
        NEXT x
    NEXT y
    RETURN total / (w * h * 3)
ENDFUNC

' Whether the region looks like the picture, within the tolerance.
EXPORT FUNC MATCHES(shot, ref, region, tolerance)
    RETURN REGION_DIFF(shot, ref, region) <= tolerance
ENDFUNC

```

```

' Carries out one step once; answers "" or why it did not work.
FUNC Attempt(s, window$, shot$)
    DIM do$ = s{"do"}
    IF do$ = "wait_window" THEN
        DIM hwnd = INPUTKIT.WAITWINDOW(window$, s{"timeout_s"} * 1000)
        IF hwnd = 0 THEN RETURN "the window did not come"
        DIM front = INPUTKIT.ACTIVATE(window$)
        IF NOT front THEN RETURN "it stayed behind"
        RETURN ""
    ENDIF
    IF do$ = "pause" THEN
        SLEEP s{"ms"}
        RETURN ""
    ENDIF
    DIM win = INPUTKIT.FINDWINDOW(window$)
    IF win = 0 THEN RETURN "the window is gone"
    IF NOT INPUTKIT.ACTIVATE(window$) THEN RETURN "it stayed behind"
    IF do$ = "check" THEN
        IF OS.SCREENSHOT(shot$, "client", window$) <> 0 THEN
            RETURN "no screenshot of the window"
        ENDIF
        DIM shot = IMG.READPNG(shot$)
        DIM ref = IMG.READPNG(s{"image"})
        DIM d = REGION_DIFF(shot, ref, s{"region"})
        IF d > s{"tolerance"} THEN
            RETURN "the region differs by " + FORMAT$("{:.1f}", d)
        ENDIF
    ELSEIF do$ = "click" ORELSE do$ = "doubleclick" THEN
        DIM p = SCREEN_POINT(INPUTKIT.CLIENT_ORIGIN(win), s{"at"})
        DIM n = 1
        IF do$ = "doubleclick" THEN n = 2
        INPUTKIT.CLICK(p[0], p[1], "left", n)
    ELSEIF do$ = "type" THEN
        INPUTKIT.TYPETEXT(s{"text"})
    ELSEIF do$ = "key" THEN
        INPUTKIT.PRESS(s{"keys"}, FALSE)
    ENDIF
    RETURN ""
ENDFUNC

' Runs the job: each step up to its tries, a second between tries, a
' log line for each. The emergency stop of INPUTKIT ends it at once.
' Answers {"ok": TRUE or FALSE, "step": the step that failed, "why"}.
EXPORT FUNC RUN(job, lg, shots_dir$)
    DIM steps = job{"steps"}
    DIM window$ = job{"window"}
    DIM shot$ = PATH.JOIN$(shots_dir$, "robot_check.png")
    DIM k = 0

```

```

FOR k = 0 TO LEN(steps) - 1
    DIM s = steps[k]
    DIM what$ = PLAN_LINE$(s, window$)
    DIM why$ = ""
    DIM try_no = 0
    DO
        try_no = try_no + 1
        TRY
            why$ = Attempt(s, window$, shot$)
        CATCH
            why$ = ERRMSG$
            IF INSTR(why$, "stopped") >= 0 THEN
                LOGGER.WARN(lg, what$ + ": " + why$)
                DIM stop_at = s{"line"}
                RETURN {"ok": FALSE, "step": stop_at, "why": why$}
            ENDIF
        ENDTRY
        IF why$ = "" THEN
            LOGGER.INFO(lg, what$ + ": ok")
            EXITDO
        ENDIF
        DIM said$ = what$ + ": try " + STR$(try_no) + ": " + why$
        LOGGER.WARN(lg, said$)
        IF try_no >= s{"tries"} THEN
            RETURN {"ok": FALSE, "step": s{"line"}, "why": why$}
        ENDIF
        SLEEP 1000
    LOOP
NEXT k
RETURN {"ok": TRUE, "step": 0, "why": ""}
ENDFUNC

```

The emergency stop lives in the shared module `inputkit.jdb`. Before every single action it asks Windows for the state of Esc and the position of the mouse:

```

EXPORT FUNC SAFE$(esc_down, x, y)
    IF esc_down THEN RETURN "Esc is held: stopped."
    IF x <= CORNER_PX AND ALSO y <= CORNER_PX THEN
        RETURN "The mouse is in the top-left corner: stopped."
    ENDIF
    RETURN ""
ENDFUNC

```

```

EXPORT SUB GUARD()
    DIM esc = KEYDOWN(GetAsyncKeyState(VK_ESCAPE))
    DIM at = CURSOR_AT()

```



```

    DIM why$ = SAFE$(esc, at[0], at[1])
    IF why$ <> "" THEN THROW "INPUTKIT: " + why$
ENDSUB

```

How it works

1. `ROBOT.LOAD` reads the job and checks every step: a kind it does not know, a region without four numbers, a missing picture or a key that closes a window, such as `Alt+F4`, is a mistake, and a job with a mistake does not run.
2. `wait_window` waits for a window with exactly the job's title, as long as `timeout_s` says, and brings it to the front.
3. Every check, click, text and key first finds the window again and brings it to the front, so a dialog that popped up in between cannot catch the robot's key strokes.
4. `check` takes a picture of the window's inside with `OS.SCREENSHOT` and compares the region with the reference picture. `REGION_DIFF` answers the mean difference of the red, green and blue values, from 0 for the same picture to 255; a check passes up to its tolerance. A region outside the window or a picture of another size never passes.
5. `click` adds the window's position on the screen to the point of the job, so the click lands in the same place of the program wherever the window is.
6. A step that does not work is tried again after a second, up to its `tries`. A check with five tries waits up to five seconds for a slow dialog. When the tries are used up, the job ends.
7. Every try writes a line to `logs/desktop_robot.log`. The program ends with exit code 1 when the job stopped, which is what the job server of X01 counts as a failed run.

Run it

Look at the job first:

```

jdbasic desktop_robot.jdb --dry-run
Job order_entry.toml for 'Order Entry - Version 4':
  step 1: wait up to 60 s for 'Order Entry - Version 4'
  step 2: check 96 x 24 at 12, 44 against new_button.png, 5 tries
  step 3: click at 60, 56

```

```

step 4: pause 800 ms
step 5: type 'Müller GmbH'
step 6: press tab
step 7: type 'Copy paper, 10 packs'
step 8: press ctrl+s, 2 tries
8 steps. Without --dry-run the robot runs them.

```

Then open the program, start the robot without `--dry-run`, and watch the first runs with your hands off the keyboard. `--job` runs another job file, such as `--job ~/Documents/AutomateWork/robot/booking.toml`.

Schedule it

Under the job server of X01, the robot is a job like any other. Add a block to `jobs.toml`:

```

[[job]]
name = "orders robot"
when = "weekdays 12:30"
program = "X16_desktop_robot/desktop_robot.jdb"
attempts = 1
catch_up = false

```

One attempt, because a job that stopped halfway has entered half an order; a second run would enter it again. `catch_up = false`, because a robot should not start on its own when you come back to a computer that was off at half past twelve. A failed run lands in the job state, where the health checks of X13 find it and, with their alerts on, tell you by mail.

Make it yours

The settings are the `[desktop_robot]` part of `work.conf`:

```

[desktop_robot]
job = "~/Documents/AutomateWork/robot/order_entry.toml"
countdown = 5

```

- **A job per chore.** Keep one job file per chore next to its pictures, and start the others with `--job`.
-

Check before every click that matters. A check step costs a second and saves you from a click on the wrong button. Put one before every click that changes data.

- **Tolerance.** A tolerance of 10 to 15 allows for a blinking cursor or a different colour of the window frame. If a check fails with a difference of 2 or 3, raise it; if it passes on the wrong screen, lower it and pick a region with more detail.
- **Rows from a sheet.** For many rows, build the job's text from the rows of a sheet and hand it to `ROBOT.PARSE_JOB`, one job per row, with a check at the start of each.

When it goes wrong

- **"the region differs by 41.3":** the program looks different from the reference picture. The display scaling, the Windows theme or a new version of the program changes every pixel; take the reference picture again.
- **"the window did not come":** the program was not open within `timeout_s`, or its title changed. Check the title bar letter by letter.
- **"it stayed behind":** Windows kept another window in front, which it does when the screen is locked or another program demands attention. Look at the screen and run the job again.
- **The job stopped halfway:** the log names the last step that worked. Finish that entry by hand before you start the robot again; the robot has no undo.

Balance dividend

About 60 minutes a week for forty corrections, entered the same way, with a log that shows each one.

Bonus: X17 Meeting Briefing

The chore

Lena has four to six meetings a day. Before each one she opens the invitation to see who comes, searches her mail for what these people wrote last, scrolls through her notes for what she promised them, and tries to remember what the meeting was for. Five minutes per meeting on a good day, and on a bad day she walks in without having done it and spends the first ten minutes catching up.

What you get

Every morning at half past seven a PDF waits in your work folder with one page per meeting of the day: the time and the room, the people, the latest mails with them, your open items that name them, and a few sentences written by a language model that sum up what the meeting is about and what the others are waiting for.

Budget review Q3

2026-10-05, 09:00 to 10:00, Room 3

People

Anna Berg, Tom Weber

In short

Anna wants the Q3 figures with the travel line split out, and Tom has put the September training invoices into the shared folder. Bring the sheet and ask Tom about the training costs.

Latest mails

2026-10-02 10:05 Tom Weber: Training costs

2026-10-01 09:15 Anna Berg: Q3 figures for the review

Open items

[] Send Anna the Q3 sheet with the travel line

[] Budget review Q3: ask Tom about the training costs

One page of the morning briefing

Before you start

The recipe reads three things you probably have already:

- **Your calendar as an .ics file.** Outlook exports one under *File, Save Calendar*; many calendar services offer a private address of the calendar in this format. The Calendar Bridge of X05 writes one as well.
- **Your saved mails** as .eml files, the folder the Inbox Assistant (X02) reads, and the outbox/sent folder of Chapter 4.
- **Your notes file** with open items as - [] lines, the one the Personal Dashboard (M08) reads.

For the summary you need a language model, as for X02: a key in OPENAI_API_KEY or ANTHROPIC_API_KEY, or a model server on your own computer such as Ollama, named in base_url. Without either, the briefing is written as well, only without the sentences on top. The recipe never asks for a key, because it runs at half past seven when nobody is there to type one.

Watch out

The summary means that the mails and notes of a meeting go to the language model. A mail you hand to a service goes to that service. Before you use a service for work mail, ask whether your company allows it. Mails with personal data or business secrets are a reason to use a model on your own computer, or to switch the summary off with `use_model = false`.

The program

The program reads the settings, finds the day's meetings and builds one page per meeting:

```
' Meeting Briefing: every morning one page per meeting of the day,
' with the people, the latest mails with them, the open items of your
' notes and a short summary by a language model, as a PDF in your
' work folder. The settings are the [meeting_briefing] part of
' work.conf.
'
' jdbasic meeting_briefing.jdb                today's meetings
' jdbasic meeting_briefing.jdb --day 2026-10-05  another day
' jdbasic meeting_briefing.jdb --dry-run        show, ask nothing
' jdbasic meeting_briefing.jdb --no-model       without a summary
```

```

'    jdbcbasic meeting_briefing.jdb --send                send the outbox

IMPORT WORKCONF, BRIEFING, OUTBOX, MAIL, LOGGER

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_ALL(cfg, OUTBOX.ASK$(cfg))
    PRINT result{"sent"}; " sent"
END
ENDIF

DIM key$ = "meeting_briefing."
DIM home$ = "~/Documents/AutomateWork"
DIM ics_home$ = home$ + "/calendar.ics"
DIM ics$ = WORKCONF.FOLDER$(cfg, key$ + "calendar", ics_home$)
DIM todo$ = home$ + "/todo.txt"
DIM notes$ = WORKCONF.FOLDER$(cfg, key$ + "notes", todo$)
DIM out_home$ = home$ + "/briefings"
DIM folder$ = WORKCONF.FOLDER$(cfg, key$ + "folder", out_home$)
DIM saved = [home$ + "/saved mail"]
DIM listed = WORKCONF.VALUE(cfg, key$ + "mail_folders", saved)
DIM folders = SELECT(LAMBDA s$ -> WORKCONF.EXPAND$(s$), listed)
PUSH(folders, PATH.JOIN$(OUTBOX.FOLDER$(cfg), "sent"))
DIM most = WORKCONF.VALUE(cfg, key$ + "mails_per_meeting", 3)
DIM use_model = WORKCONF.VALUE(cfg, key$ + "use_model", TRUE)
DIM to_self = WORKCONF.VALUE(cfg, key$ + "mail_to_self", FALSE)
DIM day$ = WORKCONF.ARG$("day")
IF day$ = "" THEN day$ = FORMAT_DATE(NOW(), "%Y-%m-%d")
DIM dry_run = WORKCONF.FLAG("dry-run")
IF WORKCONF.FLAG("no-model") THEN use_model = FALSE

DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)
DIM log_dir$ = PATH.JOIN$(work$, "logs")
MKDIR log_dir$
DIM lg = LOGGER.NEW("meeting_briefing")
LOGGER.TO_FILE(lg, PATH.JOIN$(log_dir$, "meeting_briefing.log"))

DIM meetings = BRIEFING.MEETINGS(ics$, day$)
IF LEN(meetings) = 0 THEN
    PRINT "No meetings on "; day$
    LOGGER.INFO(lg, "no meetings on " + day$)
END
ENDIF

DIM provider$ = WORKCONF.VALUE(cfg, key$ + "provider", "openai")
DIM model$ = WORKCONF.VALUE(cfg, key$ + "model", "gpt-4o-mini")
DIM base$ = WORKCONF.VALUE(cfg, key$ + "base_url", "")
DIM has_key = GETENV$("OPENAI_API_KEY") <> "" ORELSE _

```

```

    GETENV$("ANTHROPIC_API_KEY") <> ""
IF use_model AND ALSO base$ = "" AND ALSO NOT has_key THEN
    PRINT "No API key in the environment: no summaries this time."
    LOGGER.WARN(lg, "no API key, writing without summaries")
    use_model = FALSE
ENDIF
DIM ai AS MAP
IF use_model AND ALSO NOT dry_run THEN
    ai = BRIEFING.CLIENT(provider$, model$, base$, "")
ENDIF

DIM pages = []
DIM k = 0
FOR k = 0 TO LEN(meetings) - 1
    DIM m = meetings[k]
    DIM mails = BRIEFING.MAILS(folders, m, most)
    DIM items = BRIEFING.ITEMS(notes$, m)
    IF dry_run THEN
        PRINT BRIEFING.PLAN_LINE$(m, LEN(mails), LEN(items))
    ELSE
        DIM summary$ = ""
        IF use_model THEN
            TRY
                DIM prompt$ = BRIEFING.PROMPT$(m, mails, items)
                summary$ = BRIEFING.SUMMARY$(ai, prompt$)
            CATCH
                PRINT m{"title"}; ": no summary, "; ERRMSG$
                DIM fail AS MAP
                fail{"meeting"} = m{"title"}
                fail{"error"} = ERRMSG$
                LOGGER.ERROR(lg, "summary failed", fail)
            ENDTRY
        ENDIF
        DIM pg AS MAP
        pg{"meeting"} = m
        pg{"mails"} = mails
        pg{"items"} = items
        pg{"summary"} = summary$
        PUSH(pages, pg)
        PRINT m{"start"}; " "; m{"title"}
    ENDIF
NEXT k
IF dry_run THEN
    PRINT LEN(meetings); " meeting(s); nothing written"
END
ENDIF

MKDIR folder$

```

```

DIM pdf$ = PATH.JOIN$(folder$, "briefing " + day$ + ".pdf")
BRIEFING.WRITE_PDF(pdf$, day$, pages)
PRINT "wrote "; pdf$
DIM done AS MAP
done{"day"} = day$
done{"meetings"} = LEN(pages)
LOGGER.INFO(lg, "briefing written", done)
IF to_self THEN
    DIM me$ = OUTBOX.FROM$(cfg)
    DIM msg = MAIL.MESSAGE(me$, me$, "Briefing " + day$)
    MAIL.PLAIN(msg, "Your meetings of " + day$ + " are in the PDF.")
    MAIL.ATTACH(msg, pdf$)
    DIM eml$ = "briefing " + day$ + ".eml"
    PRINT "outbox "; OUTBOX.PUT$(msg, cfg, eml$)
ENDIF

```

The module does the work, from the calendar to the PDF:

```

' briefing.jdb: the work of the Meeting Briefing. It reads the day's
' meetings from a calendar file, finds the latest mails with the
' people who attend and the open items of your notes that name them,
' asks a language model for a short summary, and sets one page per
' meeting into a PDF.
'
'   IMPORT BRIEFING
'   DIM meetings = BRIEFING.MEETINGS("calendar.ics", "2026-10-05")
'   DIM mails = BRIEFING.MAILS(["C:/mail"], meetings[0], 3)
'   DIM items = BRIEFING.ITEMS("todo.txt", meetings[0])

EXPORT MODULE BRIEFING

IMPORT ICAL, DF, MAIL, LLMAPI, RETRY, PDFGEN

DIM gLF$ = CHR$(10)
DIM gMonths$ = "JanFebMarAprMayJunJulAugSepOctNovDec"

' The names of the people in a calendar file, from the CN parameter
' of its ATTENDEE and ORGANIZER lines: lower case address to name.
EXPORT FUNC NAMES(ics_text$)
    DIM out AS MAP
    DIM flat$ = REPLACE$(ics_text$, CHR$(13), "")
    flat$ = REPLACE$(flat$, gLF$ + " ", "")
    flat$ = REPLACE$(flat$, gLF$ + CHR$(9), "")
    DIM rows = SPLIT(flat$, gLF$)
    DIM k = 0
    FOR k = 0 TO LEN(rows) - 1
        DIM row$ = rows[k]

```



```

    DIM up$ = UCASE$(row$)
    DIM person = STARTSWITH(up$, "ATTENDEE")
    IF STARTSWITH(up$, "ORGANIZER") THEN person = TRUE
    DIM at = INSTR(up$, "MAILTO:")
    DIM cn = INSTR(up$, "CN=")
    IF person AND ALSO at > 0 AND ALSO cn > 0 AND ALSO cn < at THEN
        DIM mail$ = LCASE$(TRIM$(MID$(row$, at + 7)))
        DIM rest$ = MID$(row$, cn + 3, at - cn - 3)
        DIM cut = INSTR(rest$, ";")
        IF cut < 0 THEN cut = INSTR(rest$, ":")
        IF cut >= 0 THEN rest$ = LEFT$(rest$, cut)
        out{mail$} = TRIM$(REPLACE$(rest$, CHR$(34), ""))
    ENDIF
NEXT k
RETURN out
ENDFUNC

' The clock time of an instant as HH:MM, on this computer's
' clock. A time the file writes with a zone (Z or TZID) is an
' instant; a time without one names a wall clock, which ICAL
' reads in this zone, so this clock gives back the digits the
' file writes.
FUNC Clock$(t_in)
    RETURN FORMAT_DATE(CDBL(t_in), "%H:%M")
ENDFUNC

' The date an instant falls on, as YYYY-MM-DD.
FUNC Day$(t_in)
    RETURN FORMAT_DATE(CDBL(t_in), "%Y-%m-%d")
ENDFUNC

' The people of an event as maps with name and mail.
FUNC People(cal, ev, names)
    DIM who = ICAL.VALUES(cal, ev, "ATTENDEE")
    RETURN SELECT(LAMBDA USE(names) w$ -> Person(w$, names), who)
ENDFUNC

' One ATTENDEE value as a map with name and mail.
FUNC Person(w$, names)
    DIM addr$ = LCASE$(w$)
    IF STARTSWITH(addr$, "mailto:") THEN addr$ = MID$(addr$, 7)
    DIM shown$ = addr$
    IF MAP.EXISTS(names, addr$) THEN shown$ = names{addr$}
    RETURN {"name": shown$, "mail": addr$}
ENDFUNC

' The meetings of a day (YYYY-MM-DD) in a calendar file, in order of
' their start: maps with title, start, end, location and people, a

```

```

' list of maps with name and mail. Days blocked whole are left out.
EXPORT FUNC MEETINGS(ics_path$, day$)
  IF NOT FILE.EXISTS(ics_path$) THEN
    THROW "No calendar file at " + ics_path$
  ENDIF
  DIM text$ = TXTREADER$(ics_path$)
  DIM names = NAMES(text$)
  DIM cal = ICAL.PARSE(text$)
  DIM noon = CDATE(day$ + " 12:00:00")
  DIM t_from = CDBL(DATEADD("D", -1, noon))
  DIM t_to = CDBL(DATEADD("D", 1, noon))
  DIM frame = ICAL.EXPAND(cal, t_from, t_to)
  DIM found = []
  DIM keys = []
  DIM k = 0
  FOR k = 0 TO DF.NROWS(frame) - 1
    DIM r = DF.ROW(frame, k)
    DIM ev = r{"event"}
    DIM today = Day$(r{"start"}) = day$
    IF today ANDALSO NOT ICAL.ALLDAY(cal, ev) THEN
      DIM m AS MAP
      m{"title"} = r{"summary"}
      m{"start"} = Clock$(r{"start"})
      m{"end"} = Clock$(r{"end"})
      m{"location"} = ICAL.PROP$(cal, ev, "LOCATION", "")
      m{"people"} = People(cal, ev, names)
      PUSH(found, m)
      PUSH(keys, m{"start"} + CHR$(1) + STR$(LEN(found) - 1))
    ENDIF
  NEXT k
  RETURN InOrder(found, keys, FALSE, LEN(found))
ENDFUNC

' The items in the order of their keys (each ending in CHR$(1) and
' the item's number), newest first when backwards, at most most.
FUNC InOrder(items, keys, backwards, most)
  DIM order = SORT(keys)
  IF backwards THEN order = REVERSE(order)
  DIM picked = TAKE(most, order)
  RETURN SELECT(LAMBDA USE(items) key$ -> _
    items[ItemNo(key$)], picked)
ENDFUNC

' The item number at the end of a key.
FUNC ItemNo(key$)
  RETURN CINT(MID$(key$, INSTR(key$, CHR$(1)) + 1))
ENDFUNC

```

```

' The mail addresses of a meeting's people, lower case.
FUNC Addresses(meeting)
    RETURN SELECT(LAMBDA p -> p{"mail"}, meeting{"people"})
ENDFUNC

' The date header of a mail as YYYY-MM-DD HH:MM, so mails sort by it.
EXPORT FUNC SORTDATE$(date$)
    DIM parts = SPLIT(TRIM$(date$), " ")
    DIM first = 0
    IF LEN(parts) > 0 ANDALSO ENDSWITH(parts[0], ",") THEN first = 1
    IF LEN(parts) < first + 4 THEN RETURN ""
    DIM d$ = RIGHT$("0" + parts[first], 2)
    DIM mon$ = parts[first + 1]
    DIM y$ = parts[first + 2]
    DIM clock$ = LEFT$(parts[first + 3], 5)
    DIM m = INSTR(gMonths$, LEFT$(mon$, 3))
    IF m < 0 THEN RETURN ""
    DIM mm$ = RIGHT$("0" + STR$(m / 3 + 1), 2)
    RETURN y$ + "-" + mm$ + "-" + d$ + " " + clock$
ENDFUNC

' Whether a parsed mail is from or to one of the addresses.
FUNC Involves(parsed, addresses)
    DIM all = [MAIL.BAREADDRESS$(parsed{"from"})]
    DIM lists = [parsed{"to"}, parsed{"cc"}]
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(lists) - 1
        DIM one = lists[k]
        IF TYPEOF(one) = "ARRAY" THEN
            FOR j = 0 TO LEN(one) - 1
                DIM to$ = one[j]
                PUSH(all, MAIL.BAREADDRESS$(to$))
            NEXT j
        ENDIF
    NEXT k
    FOR k = 0 TO LEN(all) - 1
        DIM a$ = LCASE$(all[k])
        FOR j = 0 TO LEN(addresses) - 1
            DIM want$ = addresses[j]
            IF a$ = want$ THEN RETURN TRUE
        NEXT j
    NEXT k
    RETURN FALSE
ENDFUNC

' A parsed mail as a map with from, subject, date and text (its
' first 300 characters on one line).
```

```

FUNC Summed(parsed)
    DIM m AS MAP
    m{"from"} = parsed{"from"}
    m{"subject"} = parsed{"subject"}
    m{"date"} = SORTDATE$(parsed{"date"})
    DIM body$ = REPLACE$(TRIM$(parsed{"text"}), CHR$(13), "")
    m{"text"} = LEFT$(REPLACE$(body$, gLF$, " "), 300)
    RETURN m
ENDFUNC

' The newest mails with the meeting's people from the .eml files of
' the folders, at most most of them, as maps from Summed.
EXPORT FUNC MAILS(folders, meeting, most)
    DIM addresses = Addresses(meeting)
    DIM found = []
    DIM keys = []
    DIM f = 0
    DIM k = 0
    FOR f = 0 TO LEN(folders) - 1
        DIM folder$ = folders[f]
        IF FILE.ISDIR(folder$) THEN
            DIM names = DIR$(PATH.JOIN$(folder$, "*.eml"))
            FOR k = 0 TO LEN(names) - 1
                DIM file$ = names[k]
                DIM path$ = PATH.JOIN$(folder$, file$)
                DIM parsed = MAIL.PARSEFILE(path$)
                IF Involves(parsed, addresses) THEN
                    DIM m = Summed(parsed)
                    PUSH(found, m)
                    DIM n$ = STR$(LEN(found) - 1)
                    PUSH(keys, m{"date"} + CHR$(1) + n$)
                ENDIF
            NEXT k
        ENDIF
    NEXT f
    RETURN InOrder(found, keys, TRUE, most)
ENDFUNC

' The open items (- [ ] lines) of a notes file that name one of the
' meeting's people by first name or the meeting's title.
EXPORT FUNC ITEMS(notes_path$, meeting)
    DIM out = []
    IF NOT FILE.EXISTS(notes_path$) THEN RETURN out
    DIM firsts = SELECT(LAMBDA p -> FirstName$(p{"name"}), _
        meeting{"people"})
    DIM real = FILTER(LAMBDA s$ -> _
        LEN(s$) >= 3 AND ALSO INSTR(s$, "@") < 0, firsts)
    DIM words = APPEND([LCASE$(meeting{"title"})], real)

```

```

    DIM text$ = REPLACE$(TXTREADER$(notes_path$), CHR$(13), "")
    DIM rows = SELECT(LAMBDA s$ -> TRIM$(s$), SPLIT(text$, gLF$))
    DIM open = FILTER(LAMBDA s$ -> STARTSWITH(s$, "- [ ]"), rows)
    DIM texts = SELECT(LAMBDA s$ -> TRIM$(MID$(s$, 5)), open)
    RETURN FILTER(LAMBDA USE(words) s$ -> Mentions(s$, words), texts)
ENDFUNC

' The first name of a person, lower case.
FUNC FirstName$(name$)
    RETURN LCASE$(SPLIT(name$, " ")[0])
ENDFUNC

' Whether an item names one of the words.
FUNC Mentions(item$, words)
    DIM low$ = LCASE$(item$)
    DIM hits = FILTER(LAMBDA USE(low$) w$ -> _
        INSTR(low$, w$) >= 0, words)
    RETURN LEN(hits) > 0
ENDFUNC

' The names of a meeting's people, joined by commas.
FUNC NameList$(meeting)
    DIM names = SELECT(LAMBDA p -> p{"name"}, meeting{"people"})
    RETURN JOIN(names, ", ")
ENDFUNC

' What the language model reads about one meeting.
EXPORT FUNC PROMPT$(meeting, mails, items)
    DIM p$ = "Meeting: " + meeting{"title"} + ", " + meeting{"start"}
    p$ = p$ + " to " + meeting{"end"} + gLF$
    p$ = p$ + "People: " + NameList$(meeting) + gLF$ + gLF$
    p$ = p$ + "Recent mails:" + gLF$
    DIM k = 0
    FOR k = 0 TO LEN(mails) - 1
        DIM m = mails[k]
        p$ = p$ + "- " + m{"date"} + " from " + m{"from"} + ": "
        p$ = p$ + m{"subject"} + ". " + m{"text"} + gLF$
    NEXT k
    p$ = p$ + gLF$ + "Open items:" + gLF$
    RETURN p$ + JOIN(SELECT(LAMBDA s$ -> "- " + s$ + gLF$, items), "")
ENDFUNC

' The instructions the model gets with every meeting.
EXPORT FUNC RULES$()
    DIM r$ = "You prepare a person for a meeting. From the meeting, "
    r$ = r$ + "the recent mails and the open items, write at most "
    r$ = r$ + "four short sentences: what the meeting is about, what "
    r$ = r$ + "the others are waiting for, and what to bring. Only "

```

```

    r$ = r$ + "use what the text says; do not invent dates or "
    RETURN r$ + "numbers."
ENDFUNC

' The language model client: provider, model and base URL from the
' settings; an empty key$ leaves the key to the environment.
EXPORT FUNC CLIENT(provider$, model$, base$, key$)
    DIM ai = LLMAPI.NEW(provider$, model$, key$, base$)
    LLMAPI.SET(ai, "temperature", 0.2)
    LLMAPI.SYSTEM(ai, RULES$())
    RETURN ai
ENDFUNC

' The model's summary of a prompt; a busy or unreachable server is
' asked again up to three times.
EXPORT FUNC SUMMARY$(ai, prompt$)
    DIM why$ = ""
    DIM tries = 0
    FOR tries = 1 TO 3
        DIM text$ = ""
        DIM got = FALSE
        TRY
            text$ = LLMAPI.ASK$(ai, prompt$)
            got = TRUE
        CATCH
            why$ = ERRMSG$
            IF NOT RETRY.TRANSIENT(why$) THEN THROW why$
        ENDTRY
        IF got THEN RETURN TRIM$(text$)
        IF tries < 3 THEN SLEEP 500 * tries
    NEXT tries
    THROW why$
ENDFUNC

' A count with its word, "1 mail" or "3 mails".
FUNC Count$(n, one$, many$)
    IF n = 1 THEN RETURN "1 " + one$
    RETURN STR$(n) + " " + many$
ENDFUNC

' One line per meeting for a dry run.
EXPORT FUNC PLAN_LINE$(meeting, n_mails, n_items)
    DIM t$ = meeting{"start"} + "-" + meeting{"end"} + " "
    t$ = t$ + meeting{"title"} + " ("
    t$ = t$ + Count$(LEN(meeting{"people"}), "person", "people")
    t$ = t$ + ", " + Count$(n_mails, "mail", "mails") + ", "
    RETURN t$ + Count$(n_items, "open item", "open items") + ")"
ENDFUNC

```

```

' A heading and a block of text at the cursor, wrapped to the page.
SUB Section(pdf, title$, text$)
    PDFGEN.USEFONT(pdf, "helvetica", "B", 11)
    PDFGEN.LN(pdf, 4)
    PDFGEN.CELL(pdf, 170, 6, title$)
    PDFGEN.LN(pdf, 6)
    PDFGEN.USEFONT(pdf, "helvetica", "", 10)
    PDFGEN.MULTICELL(pdf, 170, 5, text$)
ENDSUB

' The mails of a page, one line each.
FUNC MailText$(mails)
    IF LEN(mails) = 0 THEN RETURN "No recent mails with these people."
    RETURN JOIN(SELECT(LAMBDA ml -> MailLine$(ml), mails), gLF$)
ENDFUNC

' One mail of a page as a line.
FUNC MailLine$(ml)
    DIM who$ = MAIL.ADDRESSNAME$(ml{"from"})
    IF who$ = "" THEN who$ = ml{"from"}
    RETURN ml{"date"} + " " + who$ + ": " + ml{"subject"}
ENDFUNC

' The open items of a page, one line each.
FUNC ItemText$(items)
    IF LEN(items) = 0 THEN RETURN "No open items name this meeting."
    RETURN JOIN(SELECT(LAMBDA s$ -> "[ ] " + s$, items), gLF$)
ENDFUNC

' The briefing as a PDF: one page per meeting with its time, place,
' people, summary, mails and open items. Each entry of pages is a map
' with meeting, mails, items and summary.
EXPORT SUB WRITE_PDF(path$, day$, pages)
    DIM pdf = PDFGEN.DOC()
    PDFGEN.SETINFO(pdf, "Briefing " + day$, "Meeting Briefing")
    PDFGEN.SETFOOTER(pdf, "Briefing " + day$ + ", page 615")
    DIM k = 0
    FOR k = 0 TO LEN(pages) - 1
        DIM pg = pages[k]
        DIM m = pg{"meeting"}
        PDFGEN.ADDPAGE(pdf)
        PDFGEN.USEFONT(pdf, "helvetica", "B", 16)
        PDFGEN.TEXTAT(pdf, 20, 25, m{"title"})
        PDFGEN.USEFONT(pdf, "helvetica", "", 10)
        DIM when$ = day$ + ", " + m{"start"} + " to " + m{"end"}
        DIM place$ = m{"location"}
        IF place$ <> "" THEN when$ = when$ + ", " + place$

```

```

PDFGEN.TEXTAT(pdf, 20, 32, when$)
PDFGEN.SETDRAW(pdf, 180, 180, 180)
PDFGEN.DRAWLINE(pdf, 20, 35, 190, 35)
PDFGEN.SETXY(pdf, 20, 40)
Section(pdf, "People", NameList$(m))
DIM summary$ = pg{"summary"}
IF summary$ = "" THEN summary$ = "(written without a summary)"
Section(pdf, "In short", summary$)
Section(pdf, "Latest mails", MailText$(pg{"mails"}))
Section(pdf, "Open items", ItemText$(pg{"items"}))
NEXT k
PDFGEN.WRITEFILE(pdf, path$)
ENDSUB

```

How it works

1. MEETINGS reads the calendar with the ICAL library and asks it for every event between the day before and the day after, repeating meetings included. It keeps those that start on the day you asked for and leaves out events that block whole days, such as a holiday.
2. A time in the calendar can carry a time zone or not. A time with a zone names an instant and is shown on your computer's clock. A time without one names a wall clock, which the ICAL library reads in your zone, so Clock\$ gives back the digits the file writes, which is what Outlook means by such a time.
3. NAMES reads the names of the people from the CN part of the ATTENDEE lines, so the page says "Anna Berg" and not her address.
4. MAILS reads every saved mail and keeps those from or to one of the people, newest first. SORTDATE\$ turns the date line of a mail into a text that sorts.
5. ITEMS keeps the open items of your notes that name the meeting or the first name of one of its people.
6. PROMPT\$ writes all of it into one text, and SUMMARY\$ sends it to the language model with the rules of RULES\$: four sentences at most, nothing that is not in the text. A busy server is asked again up to three times; a refused key stops at once.
7. WRITE_PDF sets one page per meeting with PDFGEN. A meeting whose summary failed gets its page anyway, and the log says why.

Run it

Look at the day first. The dry run asks no model and writes nothing:

```
jdbasic meeting_briefing.jdb --day 2026-10-05 --dry-run
09:00-10:00 Budget review Q3 (2 people, 2 mails, 2 open items)
14:00-14:30 Supplier call Miller Ltd (1 person, 1 mail, 1 open item)
2 meeting(s); nothing written
```

Then for real, here with folder set to C:/Users/lena/Briefings:

```
jdbasic meeting_briefing.jdb
09:00 Budget review Q3
14:00 Supplier call Miller Ltd
wrote C:/Users/lena/Briefings/briefing 2026-10-05.pdf
```

--no-model writes the briefing without the summaries, for a day when you would rather not send anything anywhere.

Schedule it

The wizard plans the recipe for weekdays at 07:30, before the first meeting. Under the job server of X01 it is one more block in jobs.toml, with catch_up = true: a briefing written at nine for a computer that was off at half past seven is still useful.

With mail_to_self = true the briefing also goes into the outbox as a mail to yourself, with the PDF attached. Send it with --send when you want the briefing on your phone, or open the PDF straight from the work folder.

Make it yours

The settings are the [meeting_briefing] part of work.conf:

```
[meeting_briefing]
calendar = "~/Documents/AutomateWork/calendar.ics"
mail_folders = ["~/Documents/AutomateWork/saved mail"]
notes = "~/Documents/AutomateWork/todo.txt"
folder = "~/Documents/AutomateWork/briefings"
mails_per_meeting = 3
use_model = true
provider = "openai"
model = "gpt-4o-mini"
```

```
base_url = ""
mail_to_self = false
```

- **More history.** `mails_per_meeting = 6` shows more of the conversation; the model reads them all.
- **Your own rules.** Change `RULES$` in the module to ask the model for what you need, such as "end with one question I should ask".
- **Only some meetings.** Skip meetings whose title starts with "Blocker" or "Lunch" with one line in the loop of `MEETINGS`: `IF STARTSWITH(r{"summary"}, "Lunch") THEN today = FALSE.`
- **A local model.** `base_url = "http://localhost:11434/v1"` and `model = "llama3.1"` keep every mail on your computer.

When it goes wrong

- **"No calendar file at":** the export did not happen or went somewhere else. Export it again, or point calendar at it.
- **"No meetings on 2026-10-05"** although there are some: the export covers a range of days; check that it reaches today.
- **The times are an hour off:** the calendar writes times without a zone, and the program that wrote it meant another zone than yours. Export the calendar with time zones; Outlook and most calendar services write them.
- **"no summary, ..."** in the output: the model did not answer. The log in `logs/meeting_briefing.log` has the message; the page of that meeting has no summary but everything else.

Balance dividend

About 45 minutes a week: five minutes of searching before each of nine meetings, now a glance at one page.

Bonus: X18 Spreadsheet to Web App

The chore

The team's open orders live in an Excel workbook on a shared drive. Five people update it, and twice a month someone's change is gone because two of them had the file open at the same time. Somebody types "Done" instead of "done", a date as 5.10. instead of a date, and the filters in the sheet stop working. Lena has asked IT for a small tool for months; the request is still in the queue.

What you get

The workbook becomes a small app in the browser. A list with a search box, a filter for every column that holds a few fixed values, and sorting by a click on a column head. A form to change a row or add one, with a date field for dates, a number field for numbers and a list for the fixed values, so a typing mistake cannot get in. Every save goes back into the same workbook, after a copy of the version before, and every change leaves a line in a log.

Orders

Search Status [Add a row](#)

8 of 8 rows

Order	Customer	Status	Amount	Due date	
A-104	Kraus AG	waiting	3100	2026-10-10	Edit
A-107	Berg GmbH	waiting	1890	2026-10-07	Edit
A-101	Miller Ltd	open	1200.5	2026-10-05	Edit
A-106	Novak s.r.o.	open	640	2026-10-17	Edit
A-103	Miller Ltd	open	455	2026-10-15	Edit
A-108	Kraus AG	open	220	2026-10-23	Edit
A-105	Ito & Co	done	99.9	2026-09-15	Edit
A-102	Berg GmbH	done	80	2026-09-25	Edit

The list of orders, sorted by amount

Before you start

Any workbook works whose sheet has the column names in its first row. The app guesses what each column holds:

A column of	Becomes
numbers	a number field
dates, or day numbers under a head with "date" or "due"	a date field
a few values that repeat, such as open, done, waiting	a list and a filter
anything else	a text field

Columns the app must not change, such as an order number, go into the read_only setting; the form shows them but keeps them as they are.

Watch out

The app writes the sheet back with its values. Formulas, colours and column widths of that sheet are not kept, and a formula arrives as the value it last showed. Other sheets of the workbook are written back as values too. Use the app for a workbook that is a table of data, and keep the one with the formulas as it is. The app listens only on this computer, so a colleague cannot open it from theirs; X07 shows how to give them a copy of their own.

The program

The program reads the settings, opens the sheet and starts the server:

```
' Spreadsheet to Web App: one sheet of a workbook as a small app in
' the browser on this computer, with search, filters, sorting, a form
' to change a row or add one, and saving back into the workbook. The
' settings are the [sheet_app] part of work.conf.
'
'   jdbasic sheet_app.jdb           start the app
'   jdbasic sheet_app.jdb --dry-run show the columns, start nothing

IMPORT WORKCONF, SHEETAPP, JDWEB

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())
DIM key$ = "sheet_app."
DIM home$ = "~/Documents/AutomateWork"
DIM book_home$ = home$ + "/orders.xlsx"
DIM book$ = WORKCONF.FOLDER$(cfg, key$ + "workbook", book_home$)
DIM sheet$ = WORKCONF.VALUE(cfg, key$ + "sheet", "")
DIM port = WORKCONF.VALUE(cfg, key$ + "port", 8770)
DIM fixed = WORKCONF.VALUE(cfg, key$ + "read_only", [])
DIM save_home$ = home$ + "/backups"
DIM backups$ = WORKCONF.FOLDER$(cfg, key$ + "backups", save_home$)
DIM work$ = WORKCONF.FOLDER$(cfg, "paths.work", home$)

IF NOT FILE.EXISTS(book$) THEN
    PRINT "No workbook at "; book$
    END
ENDIF
SHEETAPP.OPEN(book$, sheet$, fixed, backups$)
DIM names = SHEETAPP.HEADERS()
DIM kinds = SHEETAPP.KINDS()
DIM c = 0
FOR c = 0 TO LEN(names) - 1
    DIM name$ = names[c]
```

```

    DIM kind$ = kinds[c]
    DIM shown$ = name$ + " (" + kind$ + ")"
    IF SHEETAPP.READ_ONLY(c) THEN shown$ = shown$ + ", read only"
    PRINT shown$
NEXT c
PRINT SHEETAPP.ROW_COUNT(); " rows in "; book$
IF WORKCONF.FLAG("dry-run") THEN END

DIM log_dir$ = PATH.JOIN$(work$, "logs")
MKDIR log_dir$
SHEETAPP.LOG_TO(PATH.JOIN$(log_dir$, "sheet_app.log"))
SHEETAPP.MOUNT()
IF NOT JDWEB.LISTEN(port) THEN
    PRINT "Port "; port; " is taken; set another port in [sheet_app]."
    END
ENDIF
PRINT "The app: http://localhost: "; port; "/" (Ctrl+C stops it)
HTTP.SERVER.WAIT

```

The module holds the table, builds the pages and saves the changes:

```

' sheetapp.jdb: the work of Spreadsheet to Web App. It reads one sheet
' of a workbook, guesses what each column holds, and serves the table
' as a small app on this computer: a list with search, filters and
' sorting, a form to change a row or add one, and saving back to the
' workbook with a copy of the version before.
'
'   IMPORT SHEETAPP
'   SHEETAPP.OPEN("C:/data/orders.xlsx", "", ["Id"], "C:/backups")
'   SHEETAPP.MOUNT()
'   JDWEB.SERVE(8770)

EXPORT MODULE SHEETAPP

IMPORT XLSX, JDWEB, LOGGER

DIM gQ$ = CHR$(34)
DIM gHeaders = []
DIM gRows = []
DIM gKinds = []
DIM gChoices = []
DIM gSerial = []
DIM gVersions = []
DIM gReadOnly = []
DIM gPath$ = ""
DIM gSheet$ = ""
DIM gBackups$ = ""

```

```

DIM gStamp$ = ""
DIM glog$ = ""

' — reading the sheet —————

' A text safe to put into HTML.
EXPORT FUNC ESC$(t$)
    DIM s$ = REPLACE$(t$, "&", "&amp;")
    s$ = REPLACE$(REPLACE$(s$, "<", "&lt;"), ">", "&gt;")
    RETURN REPLACE$(s$, gQ$, "&quot;")
ENDFUNC

' The state of a file on disk, to notice a change by someone else.
EXPORT FUNC STAMP$(path$)
    IF NOT FILE.EXISTS(path$) THEN RETURN ""
    DIM info = FILE.STAT(path$)
    RETURN info{"mtime"} + "/" + STR$(info{"size"})
ENDFUNC

' An Excel day number as YYYY-MM-DD.
EXPORT FUNC SERIAL_DATE$(serial)
    DIM base = CDATE("1899-12-30 12:00:00")
    DIM day = DATEADD("D", INT(serial), base)
    RETURN FORMAT_DATE(day, "%Y-%m-%d")
ENDFUNC

' A YYYY-MM-DD date as an Excel day number.
EXPORT FUNC DATE_SERIAL(iso$)
    DIM base = CDATE("1899-12-30 12:00:00")
    RETURN ROUND(DATEDIFF("D", base, CDATE(iso$ + " 12:00:00")))
ENDFUNC

FUNC IsIso(t$)
    DIM iso$ = "^[0-9]{4}-[0-9]{2}-[0-9]{2}$"
    RETURN LEN(REGEX.FINDALL(iso$, t$)) > 0
ENDFUNC

FUNC IsNumber(v)
    RETURN TYPEOF(v) = "INT64" ORELSE TYPEOF(v) = "FLOAT64"
ENDFUNC

' What a column holds: "number", "date", "choice" or "text", from its
' header and its values. Day numbers under a header that says date or
' due are dates; a text column whose few values repeat is a choice.
EXPORT FUNC KIND$(header$, values)
    DIM n = 0
    DIM numbers = 0
    DIM isos = 0

```

```

DIM seen AS MAP
DIM k = 0
FOR k = 0 TO LEN(values) - 1
    DIM v = values[k]
    IF TYPEOF(v) <> "NONE" THEN
        DIM t$ = TRIM$("" + v)
        IF t$ <> "" THEN
            n = n + 1
            IF IsNumber(v) THEN numbers = numbers + 1
            IF IsIso(t$) THEN isos = isos + 1
            seen{t$} = TRUE
        ENDIF
    ENDIF
NEXT k
IF n = 0 THEN RETURN "text"
DIM low$ = LCASE$(header$)
DIM dated = INSTR(low$, "date") >= 0
IF INSTR(low$, "due") >= 0 THEN dated = TRUE
IF isos = n THEN RETURN "date"
IF numbers = n AND ALSO dated THEN RETURN "date"
IF numbers = n THEN RETURN "number"
DIM distinct = LEN(MAP.KEYS(seen))
IF n < 4 ORELSE distinct > 6 THEN RETURN "text"
IF distinct * 2 <= n + 1 THEN RETURN "choice"
RETURN "text"
ENDFUNC

' The values of one column of the rows.
FUNC ColumnValues(c)
    RETURN SELECT(LAMBDA USE(c) row -> row[c], gRows)
ENDFUNC

' The different texts of a column, sorted.
FUNC Distinct(values)
    DIM given = FILTER(LAMBDA v -> TYPEOF(v) <> "NONE", values)
    DIM texts = SELECT(LAMBDA v -> TRIM$("" + v), given)
    RETURN SORT(UNIQUE(FILTER(LAMBDA t$ -> t$ <> "", texts)))
ENDFUNC

' Reads the sheet (the first one when sheet$ is "") into the app.
' read_only names columns the form shows but does not change.
EXPORT SUB OPEN(path$, sheet$, read_only, backups$)
    DIM book = XLSX.READ(path$)
    DIM name$ = sheet$
    IF name$ = "" THEN name$ = XLSX.SHEETS(path$)[0]
    IF NOT MAP.EXISTS(book, name$) THEN
        THROW "No sheet " + name$ + " in " + path$
    ENDIF

```



```

DIM table = book{name$}
IF LEN(table) = 0 THEN THROW "The sheet " + name$ + " is empty"
gPath$ = path$
gSheet$ = name$
gBackups$ = backups$
gReadOnly = read_only
gStamp$ = STAMP$(path$)
gHeaders = []
DIM head = table[0]
DIM c = 0
FOR c = 0 TO LEN(head) - 1
    PUSH(gHeaders, TRIM$("" + head[c]))
NEXT c
gRows = DROP(1, table)
gKinds = []
gChoices = []
gSerial = []
FOR c = 0 TO LEN(gHeaders) - 1
    DIM values = ColumnValues(c)
    DIM kind$ = KIND$(gHeaders[c], values)
    PUSH(gKinds, kind$)
    DIM numeric = FALSE
    IF kind$ = "date" AND ALSO LEN(values) > 0 THEN
        numeric = IsNumber(values[0])
    ENDIF
    PUSH(gSerial, numeric)
    DIM opts = []
    IF kind$ = "choice" THEN opts = Distinct(values)
    PUSH(gChoices, opts)
NEXT c
gVersions = ONES(LEN(gRows))
ENDSUB

' The sheet's column names, kinds and number of rows.
EXPORT FUNC HEADERS()
    RETURN gHeaders
ENDFUNC

EXPORT FUNC KINDS()
    RETURN gKinds
ENDFUNC

EXPORT FUNC ROW_COUNT()
    RETURN LEN(gRows)
ENDFUNC

' A cell as the app shows it.
EXPORT FUNC SHOWN$(r, c)

```

```

    DIM row = gRows[r]
    IF c >= LEN(row) THEN RETURN ""
    DIM v = row[c]
    IF TYPEOF(v) = "NONE" THEN RETURN ""
    IF gKinds[c] = "date" ANDALSO gSerial[c] ANDALSO IsNumber(v) THEN
        RETURN SERIAL_DATE$(v)
    ENDIF
    RETURN "" + v
ENDFUNC

' — the list —————

' Whether a row holds the search text in any cell and passes the
' filters, a map of column number to the value it must have.
FUNC Matches(r, q$, filters)
    DIM keys = MAP.KEYS(filters)
    DIM k = 0
    FOR k = 0 TO LEN(keys) - 1
        DIM key$ = keys[k]
        DIM want$ = filters{key$}
        DIM have$ = SHOWN$(r, CINT(key$))
        IF want$ <> "" ANDALSO have$ <> want$ THEN RETURN FALSE
    NEXT k
    IF q$ = "" THEN RETURN TRUE
    DIM c = 0
    FOR c = 0 TO LEN(gHeaders) - 1
        DIM cell$ = LCASE$(SHOWN$(r, c))
        IF INSTR(cell$, LCASE$(q$)) >= 0 THEN RETURN TRUE
    NEXT c
    RETURN FALSE
ENDFUNC

' The row numbers to show, in order: search text, filters, and the
' column to sort by (-1 for the sheet's order).
EXPORT FUNC VIEW(q$, filters, sort_col, descending)
    DIM out = FILTER(LAMBDA USE(q$, filters) r -> _
        Matches(r, q$, filters), IOTA(LEN(gRows)) - 1)
    IF sort_col < 0 THEN RETURN out
    DIM keys = SELECT(LAMBDA USE(sort_col) r -> _
        SortKey$(r, sort_col), out)
    DIM sorted = SELECT(LAMBDA s$ -> _
        CINT(MID$(s$, LEN(s$) - 6)), SORT(keys))
    IF descending THEN sorted = REVERSE(sorted)
    RETURN sorted
ENDFUNC

' The sort key of a row: the cell of the column, numbers padded so
' they sort as numbers, then the row number.

```

```

FUNC SortKey$(r, sort_col)
    DIM s$ = LCASE$(SHOWN$(r, sort_col))
    IF gKinds[sort_col] = "number" AND ALSO s$ <> "" THEN
        s$ = FORMAT$("{:020.4f}", VAL(s$) + 1000000000)
    ENDIF
    RETURN s$ + CHR$(1) + RIGHT$("000000" + STR$(r), 6)
ENDFUNC

' name="value", with the value escaped.
FUNC Attr$(name$, value$)
    RETURN " " + name$ + "=" + gQ$ + ESC$(value$) + gQ$
ENDFUNC

' An option of a list, selected when it is the one chosen.
FUNC Opt$(text$, chosen$)
    DIM sel$ = ""
    IF text$ = chosen$ THEN sel$ = " selected"
    RETURN "<option" + sel$ + ">" + ESC$(text$) + "</option>"
ENDFUNC

FUNC Style$()
    DIM s$ = "body{font:15px/1.4 Segoe UI,Arial,sans-serif;"
    s$ = s$ + "margin:24px;color:#222}"
    s$ = s$ + "table{border-collapse:collapse;margin-top:12px}"
    s$ = s$ + "th,td{border-bottom:1px solid #ddd;padding:6px 10px;"
    s$ = s$ + "text-align:left}"
    s$ = s$ + "th a{color:#1f3864;text-decoration:none}"
    s$ = s$ + "tr:hover{background:#f3f6fb}"
    s$ = s$ + ".msg{background:#eaf1fb;border-left:5px solid #2e74b5;"
    s$ = s$ + "padding:8px 12px}.err{background:#fbeaea;"
    s$ = s$ + "border-left:5px solid #b52e2e;padding:8px 12px}"
    s$ = s$ + "input,select{font:inherit;padding:4px}"
    RETURN s$ + "label{display:block;margin-top:10px}"
ENDFUNC

FUNC Page$(title$, body$)
    DIM h$ = "<!doctype html><html><head><meta charset=utf-8>"
    h$ = h$ + "<title>" + ESC$(title$) + "</title><style>"
    h$ = h$ + Style$() + "</style></head><body><h1>"
    RETURN h$ + ESC$(title$) + "</h1>" + body$ + "</body></html>"
ENDFUNC

' The list page: search box, a filter per choice column, the table
' with sortable headers, an edit link per row and a link to add one.
EXPORT FUNC LIST_HTML$(q$, filters, sort_col, descending, message$)
    DIM rows = VIEW(q$, filters, sort_col, descending)
    DIM b$ = ""
    IF message$ <> "" THEN

```

```

    b$ = "<p class=msg>" + ESC$(message$) + "</p>"
ENDIF
b$ = b$ + "<form method=get action=/>Search <input"
b$ = b$ + Attr$("name", "q") + Attr$("value", q$) + ">"
DIM c = 0
DIM k = 0
FOR c = 0 TO LEN(gHeaders) - 1
    IF gKinds[c] = "choice" THEN
        b$ = b$ + " " + ESC$(gHeaders[c]) + " <select"
        b$ = b$ + Attr$("name", "f" + STR$(c)) + ">"
        b$ = b$ + "<option value=" + gQ$ + gQ$ + ">all</option>"
        DIM chosen$ = ""
        DIM fk$ = STR$(c)
        IF MAP.EXISTS(filters, fk$) THEN chosen$ = filters{fk$}
        DIM opts = gChoices[c]
        FOR k = 0 TO LEN(opts) - 1
            b$ = b$ + Opt$(opts[k], chosen$)
        NEXT k
        b$ = b$ + "</select>"
    ENDIF
NEXT c
b$ = b$ + " <button>Show</button>"
b$ = b$ + " <a href=/new>Add a row</a></form>"
b$ = b$ + "<p>" + STR$(LEN(rows)) + " of " + STR$(LEN(gRows))
b$ = b$ + " rows</p><table><tr>"
FOR c = 0 TO LEN(gHeaders) - 1
    DIM dir$ = "asc"
    IF c = sort_col AND ALSO NOT descending THEN dir$ = "desc"
    DIM link$ = "/?sort=" + STR$(c) + "&dir=" + dir$ + "&q=" + q$
    b$ = b$ + "<th><a" + Attr$("href", link$) + ">"
    b$ = b$ + ESC$(gHeaders[c]) + "</a></th>"
NEXT c
b$ = b$ + "<th></th></tr>"
FOR k = 0 TO LEN(rows) - 1
    DIM r = rows[k]
    b$ = b$ + "<tr>"
    FOR c = 0 TO LEN(gHeaders) - 1
        b$ = b$ + "<td>" + ESC$(SHOWN$(r, c)) + "</td>"
    NEXT c
    b$ = b$ + "<td><a href=/edit/" + STR$(r) + ">Edit</a>"
    b$ = b$ + "</td></tr>"
NEXT k
RETURN Page$(gSheet$, b$ + "</table>")
ENDFUNC

```

' — the form —————

' Whether a column is one the form must not change.

```

EXPORT FUNC READ_ONLY(c)
    DIM k = 0
    FOR k = 0 TO LEN(gReadOnly) - 1
        DIM name$ = gReadOnly[k]
        IF LCASE$(name$) = LCASE$(gHeaders[c]) THEN RETURN TRUE
    NEXT k
    RETURN FALSE
ENDFUNC

' The field of column c in the form, holding the text v$.
FUNC Field$(c, v$)
    DIM nm$ = Attr$("name", "c" + STR$(c))
    IF READ_ONLY(c) THEN
        RETURN "<input" + nm$ + Attr$("value", v$) + " readonly>"
    ENDIF
    IF gKinds[c] = "choice" THEN
        DIM f$ = "<select" + nm$ + ">"
        DIM opts = gChoices[c]
        DIM k = 0
        FOR k = 0 TO LEN(opts) - 1
            f$ = f$ + Opt$(opts[k], v$)
        NEXT k
        RETURN f$ + "</select>"
    ENDIF
    DIM kind$ = "text"
    IF gKinds[c] = "number" THEN kind$ = "number step=any"
    IF gKinds[c] = "date" THEN kind$ = "date"
    RETURN "<input type=" + kind$ + nm$ + Attr$("value", v$) + ">"
ENDFUNC

' The form for row r, or for a new row when r is -1, with the values
' given (a map of column number to text) or the row's own.
EXPORT FUNC FORM_HTML$(r, given, message$)
    DIM title$ = "New row"
    IF r >= 0 THEN title$ = "Row " + STR$(r + 2)
    DIM b$ = ""
    IF message$ <> "" THEN
        b$ = "<p class=err>" + ESC$(message$) + "</p>"
    ENDIF
    DIM version = 0
    IF r >= 0 THEN version = gVersions[r]
    b$ = b$ + "<form method=post action=/save>"
    b$ = b$ + "<input type=hidden name=row value=" + STR$(r) + ">"
    b$ = b$ + "<input type=hidden name=version value="
    b$ = b$ + STR$(version) + ">"
    DIM c = 0
    FOR c = 0 TO LEN(gHeaders) - 1
        DIM v$ = ""

```

```

    IF MAP.EXISTS(given, STR$(c)) THEN
        v$ = given{STR$(c)}
    ELSEIF r >= 0 THEN
        v$ = SHOWN$(r, c)
    ENDIF
    b$ = b$ + "<label>" + ESC$(gHeaders[c]) + "</label>"
    b$ = b$ + Field$(c, v$)
NEXT c
b$ = b$ + "<p><button>Save</button> <a href=/>Back</a></p></form>"
RETURN Page$(title$, b$)
ENDFUNC

' "" when a text fits its column, else what is wrong with it.
EXPORT FUNC CHECK$(c, t$)
    IF t$ = "" THEN RETURN ""
    DIM kind$ = gKinds[c]
    DIM pattern$ = "^-?[0-9]+(.[0-9]+)?$"
    DIM fits = LEN(REGEX.FINDALL(pattern$, t$)) > 0
    IF kind$ = "number" AND ALSO NOT fits THEN
        RETURN gHeaders[c] + " needs a number, such as 12.5"
    ENDIF
    IF kind$ = "date" AND ALSO NOT IsIso(t$) THEN
        RETURN gHeaders[c] + " needs a date as YYYY-MM-DD"
    ENDIF
    IF kind$ = "choice" THEN
        DIM opts = gChoices[c]
        DIM k = 0
        FOR k = 0 TO LEN(opts) - 1
            DIM o$ = opts[k]
            IF o$ = t$ THEN RETURN ""
        NEXT k
        RETURN gHeaders[c] + " is one of " + JOIN(opts, ", ")
    ENDIF
    RETURN ""
ENDFUNC

' A text as the cell the workbook stores for its column.
FUNC Cellof(c, t$)
    IF t$ = "" THEN RETURN ""
    IF gKinds[c] = "number" THEN RETURN VAL(t$)
    IF gKinds[c] = "date" AND ALSO gSerial[c] THEN
        RETURN DATE_SERIAL(t$)
    ENDIF
    RETURN t$
ENDFUNC

' Changes row r (or adds one when r is -1) from the texts given, a map
' of column number to text. version is the row's version the form

```

```

' showed. Answers "" on success, else the reason; a row changed in
' the meantime is refused.
EXPORT FUNC APPLY$(r, version, given)
  IF r >= LEN(gRows) THEN RETURN "There is no row " + STR$(r + 2)
  IF r >= 0 AND ALSO gVersions[r] <> version THEN
    RETURN "Changed by someone else in the meantime; reopen it"
  ENDIF
  DIM c = 0
  FOR c = 0 TO LEN(gHeaders) - 1
    IF MAP.EXISTS(given, STR$(c)) THEN
      DIM why$ = CHECK$(c, given{STR$(c)})
      IF why$ <> "" THEN RETURN why$
    ENDIF
  NEXT c
  DIM row = []
  DIM changes = []
  FOR c = 0 TO LEN(gHeaders) - 1
    DIM old$ = ""
    IF r >= 0 THEN old$ = SHOWN$(r, c)
    DIM new$ = old$
    IF MAP.EXISTS(given, STR$(c)) AND ALSO NOT READ_ONLY(c) THEN
      new$ = given{STR$(c)}
    ENDIF
    IF new$ <> old$ THEN
      PUSH(changes, gHeaders[c] + ": " + old$ + " -> " + new$)
    ENDIF
    PUSH(row, Cellof(c, new$))
  NEXT c
  DIM what$ = JOIN(changes, "; ")
  IF r < 0 THEN
    PUSH(gRows, row)
    PUSH(gVersions, 1)
    Note("added row " + STR$(LEN(gRows) + 1) + ": " + what$)
  ELSE
    gRows[r] = row
    gVersions[r] = gVersions[r] + 1
    IF what$ <> "" THEN Note("row " + STR$(r + 2) + ": " + what$)
  ENDIF
  RETURN ""
ENDFUNC

' — saving —————

SUB Note(text$)
  IF gLog$ = "" THEN RETURN
  DIM lg = LOGGER.NEW("sheet_app")
  LOGGER.TO_FILE(lg, gLog$)
  LOGGER.INFO(lg, text$)

```

```

ENDSUB

' Where the app writes its log.
EXPORT SUB LOG_TO(path$)
    gLog$ = path$
ENDSUB

' Writes the sheet back into the workbook, keeping the other sheets,
' after copying the file into the backups folder. A workbook changed
' on disk by someone else is refused.
EXPORT FUNC SAVE$()
    IF STAMP$(gPath$) <> gStamp$ THEN
        RETURN "The workbook changed on disk; restart the app"
    ENDIF
    DIM book = XLSX.READ(gPath$)
    DIM names = XLSX.SHEETS(gPath$)
    MKDIR gBackups$
    DIM when$ = FORMAT_DATE(NOW(), "%Y-%m-%d %H%M%S")
    DIM copy$ = PATH.BASENAME$(gPath$) + " " + when$ + ".xlsx"
    FILE.COPY(gPath$, PATH.JOIN$(gBackups$, copy$), TRUE)
    DIM wb = XLSX.NEW()
    DIM k = 0
    FOR k = 0 TO LEN(names) - 1
        DIM name$ = names[k]
        IF name$ = gSheet$ THEN
            DIM table = [gHeaders]
            table = APPEND(table, gRows)
            DIM s = XLSX.SHEET(wb, name$, table)
        ELSE
            DIM other = XLSX.SHEET(wb, name$, book{name$})
        ENDIF
    NEXT k
    XLSX.WRITE(gPath$, wb)
    gStamp$ = STAMP$(gPath$)
    RETURN ""
ENDFUNC

' — the web routes —————

' The filters of a request: f0, f1 ... as column number to value.
FUNC FiltersOf(request)
    DIM out AS MAP
    DIM c = 0
    FOR c = 0 TO LEN(gHeaders) - 1
        DIM v$ = JDWEB.PARAM$(request, "f" + STR$(c))
        IF v$ <> "" THEN out{STR$(c)} = v$
    NEXT c
    RETURN out

```



```

ENDFUNC

FUNC OnList(request)
    DIM sort_col = -1
    DIM s$ = JDWEB.PARAM$(request, "sort")
    IF s$ <> "" THEN sort_col = CINT(s$)
    IF sort_col >= LEN(gHeaders) THEN sort_col = -1
    DIM down = (JDWEB.PARAM$(request, "dir") = "desc")
    DIM q$ = JDWEB.PARAM$(request, "q")
    DIM msg$ = JDWEB.PARAM$(request, "msg")
    DIM f = FiltersOf(request)
    RETURN JDWEB.REPLY(LIST_HTML$(q$, f, sort_col, down, msg$))
ENDFUNC

FUNC OnEdit(request)
    DIM r = CINT(VAL(JDWEB.PARAM$(request, "n")))
    IF r < 0 ORELSE r >= LEN(gRows) THEN
        RETURN JDWEB.REPLY("No such row", 404, "text/plain")
    ENDIF
    DIM blank AS MAP
    RETURN JDWEB.REPLY(FORM_HTML$(r, blank, ""))
ENDFUNC

FUNC OnNew(request)
    DIM blank AS MAP
    RETURN JDWEB.REPLY(FORM_HTML$(-1, blank, ""))
ENDFUNC

FUNC OnSave(request)
    DIM r = CINT(VAL(JDWEB.PARAM$(request, "row")))
    DIM version = CINT(VAL(JDWEB.PARAM$(request, "version")))
    DIM given AS MAP
    DIM c = 0
    FOR c = 0 TO LEN(gHeaders) - 1
        DIM v$ = TRIM$(JDWEB.PARAM$(request, "c" + STR$(c)))
        given{STR$(c)} = v$
    NEXT c
    DIM why$ = APPLY$(r, version, given)
    IF why$ = "" THEN why$ = SAVE$()
    IF why$ <> "" THEN
        DIM status = 400
        IF INSTR(why$, "meantime") >= 0 THEN status = 409
        IF INSTR(why$, "on disk") >= 0 THEN status = 409
        RETURN JDWEB.REPLY(FORM_HTML$(r, given, why$), status)
    ENDIF
    DIM back = JDWEB.REPLY("", 303)
    RETURN JDWEB.HEADER(back, "Location", "/?msg=Saved")
ENDFUNC

```

```
' Puts the app's pages on the web server: the list at /, the form at
' /edit/n and /new, and /save for the form.
EXPORT SUB MOUNT()
    JDWEB.GET("/", OnList@)
    JDWEB.GET("/edit/:n", OnEdit@)
    JDWEB.GET("/new", OnNew@)
    JDWEB.POST("/save", OnSave@)
ENDSUB
```

How it works

1. OPEN reads the workbook with XLSX and keeps the chosen sheet as column names and rows. KIND\$ looks at each column's values and decides what the column holds.
2. Excel stores a date as a day number, counted from the last day of
3. SERIAL_DATE\$ turns it into a date for the page and DATE_SERIAL back into a day number for the workbook, so Excel still shows a date after the app saved it.
4. VIEW answers the row numbers to show. FILTER keeps those that hold the search text and pass the filters, and SORT orders them by the key SortKey\$ builds from the column you clicked. Numbers sort as numbers, so 1890 comes after 455.
5. LIST_HTML\$ and FORM_HTML\$ build the pages. Every text goes through ESC\$, so a customer called "Ito & Co" stays a name and never becomes part of the page's code.
6. Every row has a version number. The form sends the version it showed, and APPLY\$ refuses a save when the row has changed since, because someone else saved it in another browser tab. That is the lock that keeps two people from overwriting each other.
7. CHECK\$ refuses a word in a number field, a date in another form and a value that is not one of a column's fixed values. The form comes back with the message and what you typed.
8. SAVE\$ first checks that nobody changed the workbook on disk since the app read it, copies it into the backups folder, and writes all sheets back. Note writes one log line per change, such as row 4: Status: open -> done.

Run it

Look at the columns first:

```
jdbasic sheet_app.jdb --dry-run
Order (text), read only
Customer (text)
Status (choice)
Amount (number)
Due date (date)
8 rows in C:/Users/lena/Documents/AutomateWork/orders.xlsx
```

Then start it and open the address in your browser:

```
jdbasic sheet_app.jdb
The app: http://localhost:8770/ (Ctrl+C stops it)
```

The app runs until you close its window or press Ctrl+C. While it runs, change the workbook only through the app; the app refuses to save over a workbook someone changed in Excel and asks you to restart it.

Schedule it

The wizard sets the recipe to *by hand*: you start the app when you need it. To have it ready whenever you log on, give its `recipe.toml` a second task, as the search page of X03 does:

```
[[task]]
name = "app"
schedule = "at logon"
args = []
```

Make it yours

The settings are the `[sheet_app]` part of `work.conf`:

```
[sheet_app]
workbook = "~/Documents/AutomateWork/orders.xlsx"
sheet = ""
port = 8770
read_only = ["Order"]
backups = "~/Documents/AutomateWork/backups"
```

- **Another sheet.** `sheet = "Customers"` serves that sheet instead of the first one.
- **More fixed values.** `KIND$` treats a column with up to six values as a list when it has about twice as many rows as values. For a column with ten, raise the 6 in the line `IF n < 4 ORELSE distinct > 6 THEN RETURN "text"`.
- **Two apps at once.** Make a copy of `work.conf` whose `[sheet_app]` part names another workbook and another port, and start the second app with `--config` and that copy.
- **Old copies.** The backups folder keeps every version. Delete old copies now and then, or move the folder to a drive that is backed up anyway.

When it goes wrong

- **"Port 8770 is taken":** another program, or a second copy of the app, listens there. Close it, or set another port.
- **"Changed by someone else in the meantime; reopen it":** another tab saved the row since you opened the form. Go back to the list, open the row again, and make your change once more.
- **"The workbook changed on disk; restart the app":** someone saved the workbook in Excel while the app ran. Stop the app and start it again; it reads the new version.
- **A column is a text field although it holds numbers:** one cell of it holds text, such as "n/a". Clean the cell in Excel, then start the app again.

Balance dividend

About 40 minutes a week: no lost changes to repair, no typing mistakes to hunt in a shared sheet, and no queue at IT for a tool.

Bonus: X19 Outlook Bridge

The chore

Lena's company runs on Outlook. Her calendar is there, her mail is there, and the mail server only takes mail from Outlook, so the outbox of Chapter 4 has no server to talk to. The Meeting Briefing (X17) wants her day as an `.ics` file and her mails as `.eml` files. Each morning she saves the calendar under *File, Save Calendar* and drags the latest mails into a folder, and on the mornings she forgets, the briefing is about yesterday.

What you get

Each weekday at seven the Outlook Bridge reads today's appointments and your newest mails from Outlook and writes them where the other recipes look: the appointments as `calendar.ics`, the mails as one `.eml` file each in `saved mail`. Half an hour later the Meeting Briefing finds both. And with one line in `work.conf`, every recipe that puts mail into the outbox sends it through Outlook, from your own account, with the copy in your *Sent Items*.

Before you start

You need classic Outlook for Windows, the program that comes with Microsoft Office, set up with your account. The recipe only reads from Outlook; it never changes, moves or deletes an appointment or a mail. To send through Outlook, add one line to the `[mail]` part of `work.conf`:

```
[mail]
send_with = "outlook"
```

From then on `--send` of any recipe hands the outbox to Outlook instead of a mail server, and no password is asked for. Without the line, the outbox works as Chapter 4 describes.

Watch out

Only classic Outlook works. The new Outlook for Windows, and Outlook on the web, have no interface that other programs can use. If your Outlook has a switch *New Outlook* in the corner, it must be off. The recipe runs under your own account while you are logged in, from the Task Scheduler, not as a Windows service. Some companies set Outlook to ask before another program reads addresses or sends mail. Then Outlook shows a question the first time, and a run at seven in the morning waits for an answer nobody gives. Ask IT whether that warning is on before you schedule the recipe. On a computer of your own with an up to date virus scanner, Outlook usually asks nothing: the test send for this book went through without a question, attachment, umlauts and euro sign included.

The program

The program sends the outbox with `--send`; otherwise it reads the calendar and the inbox and writes the files:

```
' Outlook Bridge: reads the appointments of the next days and the
' newest mails from classic Outlook and writes them as an .ics file
' and .eml files, the shapes the Meeting Briefing and other recipes
' read. With --send it sends the outbox through Outlook. The settings
' are the [outlook_bridge] part of work.conf.
'
'   jdbasic outlook_bridge.jdb           export calendar and mails
'   jdbasic outlook_bridge.jdb --dry-run show what it would do
'   jdbasic outlook_bridge.jdb --send    send the outbox by Outlook

IMPORT WORKCONF, OLBRIDGE, OUTBOX

DIM cfg = WORKCONF.SETTINGS(WORKCONF.CONFIG_PATH$())

IF WORKCONF.FLAG("send") THEN
    DIM result = OUTBOX.SEND_OUTLOOK(cfg)
    PRINT result{"sent"}; " sent through Outlook"
    DIM f = 0
    FOR f = 0 TO LEN(result{"failed"}) - 1
        PRINT "  not sent: "; result{"failed"}[f]
    NEXT f
    END
ENDIF

DIM key$ = "outlook_bridge."
```

```

DIM ics_home$ = "~/Documents/AutomateWork/calendar.ics"
DIM ics$ = WORKCONF.FOLDER$(cfg, key$ + "calendar_file", ics_home$)
DIM mail_home$ = "~/Documents/AutomateWork/saved mail"
DIM mail_key$ = key$ + "mail_folder"
DIM mail_dir$ = WORKCONF.FOLDER$(cfg, mail_key$, mail_home$)
DIM days = WORKCONF.VALUE(cfg, key$ + "days", 1)
DIM most = WORKCONF.VALUE(cfg, key$ + "mails", 50)

DIM today$ = FORMAT_DATE(NOW(), "%Y-%m-%d")
DIM t_from = CDBL(CDATE(today$ + " 00:00:00"))
DIM t_to = CDBL(DATEADD("D", days, CDATE(today$ + " 00:00:00")))

IF WORKCONF.FLAG("dry-run") THEN
    PRINT "would read "; days; " day(s) of appointments from "; today$
    PRINT "    into "; ics$
    PRINT "would save the newest "; most; " mails of the inbox"
    PRINT "    into "; mail_dir$
END
ENDIF

DIM ol = CREATEOBJECT("Outlook.Application")
DIM events = []
DIM mails = []
TRY
    events = OLBRIDGE.EVENTS(ol, t_from, t_to)
    IF most > 0 THEN mails = OLBRIDGE.MAILS(ol, most)
CATCH
    DIM why$ = ERRMSG$
    RELEASEOBJECT(ol)
    PRINT "Outlook Bridge stopped: "; why$
END 1
ENDTRY
RELEASEOBJECT(ol)

MKDIR PATH.DIRNAME$(ics$)
TXTWRITER ics$, OLBRIDGE.ICS$(events)
PRINT LEN(events); " appointment(s) in "; ics$

MKDIR mail_dir$
DIM saved = 0
DIM k = 0
FOR k = 0 TO LEN(mails) - 1
    DIM m = mails[k]
    DIM path$ = PATH.JOIN$(mail_dir$, OLBRIDGE.FILE_NAME$(m))
    IF NOT FILE.EXISTS(path$) THEN
        TXTWRITER path$, OLBRIDGE.EML$(m)
        saved = saved + 1
    ENDIF
NEXT k

```

```
NEXT k
PRINT saved; " new mail(s) in "; mail_dir$
```

The module reads Outlook and writes the open formats:

```
' olbridge.jdb: the work of the Outlook Bridge. It reads appointments
' and the newest mails of classic Outlook and writes them in the open
' shapes the other recipes read: a calendar as an .ics file and mails
' as .eml files.
'
'   IMPORT OLBRIDGE
'   DIM ol = CREATEOBJECT("Outlook.Application")
'   DIM events = OLBRIDGE.EVENTS(ol, t_from, t_to)
'   TXTWRITER "today.ics", OLBRIDGE.ICS$(events)
'   RELEASEOBJECT(ol)

EXPORT MODULE OLBRIDGE

IMPORT ICAL, MAIL, OFFICEKIT

CONST OL_CALENDAR = 9
CONST OL_INBOX = 6
CONST OL_MAIL = 43
CONST OL_RESOURCE = 3
CONST MOST_ITEMS = 100000

' The appointments of the default calendar that overlap the time
' from t_from to t_to, repeating ones included, in order of start:
' maps with title, start, end, location, all_day and people (maps
' with name and mail). Outlook is only read.
EXPORT FUNC EVENTS(ol, t_from, t_to)
    DIM pattern$ = OFFICEKIT.SHORT_DATE$()
    DIM q$ = "[Start] < '" + OFFICEKIT.LOCAL_DATE$(t_to, pattern$)
    q$ = q$ + "' AND [End] > '"
    q$ = q$ + OFFICEKIT.LOCAL_DATE$(t_from, pattern$) + "'"
    DIM ns = ol.GetNamespace("MAPI")
    DIM folder = ns.GetDefaultFolder(OL_CALENDAR)
    DIM items = folder.Items
    items.Sort("[Start]")
    items.IncludeRecurrences = TRUE
    DIM found = items.Restrict(q$)
    DIM out = []
    DIM it = found.GetFirst()
    DIM n = 0
    FOR n = 1 TO MOST_ITEMS
        IF TYPEOF(it) = "NONE" THEN EXITFOR
        DIM ev AS MAP
```



```

ev{"title"} = "" + it.Subject
ev{"start"} = OFFICEKIT.PARSE_DATE(it.Start, pattern$)
ev{"end"} = OFFICEKIT.PARSE_DATE(it.End, pattern$)
ev{"location"} = "" + it.Location
ev{"all_day"} = it.AllDayEvent
DIM people = []
DIM rcpts = it.Recipients
DIM k = 0
FOR k = 1 TO rcpts.Count
    DIM r = rcpts.Item(k)
    DIM addr$ = "" + r.Address
    IF INSTR(addr$, "@") < 0 THEN
        TRY
            DIM entry = r.AddressEntry
            DIM user = entry.GetExchangeUser()
            IF TYPEOF(user) <> "NONE" THEN
                addr$ = "" + user.PrimarySmtAddress
                RELEASEOBJECT(user)
            ENDIF
            RELEASEOBJECT(entry)
        CATCH
            addr$ = ""
        ENDTRY
    ENDIF
    DIM who$ = "" + r.Name
    DIM kind = r.Type
    IF kind <> OL_RESOURCE ANDALSO INSTR(addr$, "@") > 0 THEN
        PUSH(people, {"name": who$, "mail": LCASE$(addr$)})
    ENDIF
    RELEASEOBJECT(r)
NEXT k
RELEASEOBJECT(rcpts)
ev{"people"} = people
IF ev{"start"} < t_to ANDALSO ev{"end"} > t_from THEN
    PUSH(out, ev)
ENDIF
RELEASEOBJECT(it)
it = found.GetNext()
NEXT n
RELEASEOBJECT(found)
RELEASEOBJECT(items)
RELEASEOBJECT(folder)
RELEASEOBJECT(ns)
RETURN out
ENDFUNC

```

' The newest mails of the inbox, at most most of them: maps with
' from, to, cc (lists of addresses), subject, text and received.

```

' Outlook is only read; meeting requests and reports are left out.
EXPORT FUNC MAILS(ol, most)
    DIM pattern$ = OFFICEKIT.SHORT_DATE$()
    DIM ns = ol.GetNamespace("MAPI")
    DIM folder = ns.GetDefaultFolder(OL_INBOX)
    DIM items = folder.Items
    items.Sort("[ReceivedTime]", TRUE)
    DIM out = []
    DIM it = items.GetFirst()
    DIM n = 0
    FOR n = 1 TO items.Count
        IF TYPEOF(it) = "NONE" ORELSE LEN(out) >= most THEN EXITFOR
        IF it.Class = OL_MAIL THEN
            DIM m AS MAP
            DIM from$ = "" + it.SenderEmailAddress
            IF INSTR(from$, "@") < 0 THEN
                TRY
                    DIM sender = it.Sender
                    DIM boss = sender.GetExchangeUser()
                    IF TYPEOF(boss) <> "NONE" THEN
                        from$ = "" + boss.PrimarySmtpAddress
                        RELEASEOBJECT(boss)
                    ENDIF
                    RELEASEOBJECT(sender)
                CATCH
                    from$ = ""
                ENDTRY
            ENDIF
            m{"from"} = PERSON$("" + it.SenderName, from$)
            DIM to_list = []
            DIM cc_list = []
            DIM rcpts = it.Recipients
            DIM k = 0
            FOR k = 1 TO rcpts.Count
                DIM r = rcpts.Item(k)
                DIM addr$ = "" + r.Address
                IF INSTR(addr$, "@") < 0 THEN
                    TRY
                        DIM entry = r.AddressEntry
                        DIM user = entry.GetExchangeUser()
                        IF TYPEOF(user) <> "NONE" THEN
                            addr$ = "" + user.PrimarySmtpAddress
                            RELEASEOBJECT(user)
                        ENDIF
                        RELEASEOBJECT(entry)
                    CATCH
                        addr$ = ""
                    ENDTRY
                ENDIF
            ENDIF
        ENDIF
    NEXT n
    RETURN out

```

```

        ENDIF
        DIM one$ = PERSON$(" " + r.Name, addr$)
        IF INSTR(addr$, "@") > 0 AND ALSO r.Type = 1 THEN
            PUSH(to_list, one$)
        ENDIF
        IF INSTR(addr$, "@") > 0 AND ALSO r.Type = 2 THEN
            PUSH(cc_list, one$)
        ENDIF
        RELEASEOBJECT(r)
    NEXT k
    RELEASEOBJECT(rcpts)
    m{"to"} = to_list
    m{"cc"} = cc_list
    m{"subject"} = " " + it.Subject
    m{"text"} = " " + it.Body
    DIM came$ = " " + it.ReceivedTime
    m{"received"} = OFFICEKIT.PARSE_DATE(came$, pattern$)
    PUSH(out, m)
ENDIF
RELEASEOBJECT(it)
it = items.GetNext()
NEXT n
RELEASEOBJECT(items)
RELEASEOBJECT(folder)
RELEASEOBJECT(ns)
RETURN out
ENDFUNC

' A name and an address as one mail address: "Name <addr>", or the
' bare address when there is no name.
EXPORT FUNC PERSON$(name$, addr$)
    DIM shown$ = TRIM$(REPLACE$(name$, CHR$(34), ""))
    DIM bare$ = LCASE$(TRIM$(addr$))
    IF shown$ = "" OR ELSE shown$ = bare$ THEN RETURN bare$
    IF bare$ = "" THEN RETURN ""
    RETURN CHR$(34) + shown$ + CHR$(34) + " <" + bare$ + ">"
ENDFUNC

' Events from EVENTS as the text of an .ics file. Each person becomes
' an ATTENDEE line with the name in CN, as calendars write it.
EXPORT FUNC ICS$(events)
    DIM cal = ICAL.NEW("-//AutomateWork//Outlook Bridge//EN")
    DIM k = 0
    DIM j = 0
    FOR k = 0 TO LEN(events) - 1
        DIM e = events[k]
        DIM t0 = CDBL(e{"start"})
        DIM t1 = CDBL(e{"end"})
    
```

```

    DIM ev = 0
    IF e{"all_day"} THEN
        DIM days = MAX([1, ROUND((t1 - t0) / 86400)])
        ev = ICAL.ADDALLDAY(cal, e{"title"}, t0 + 43200, days)
    ELSE
        ev = ICAL.ADDEVENT(cal, e{"title"}, t0, t1)
    ENDIF
    IF e{"location"} <> "" THEN
        ICAL.SETTEXT(cal, ev, "LOCATION", e{"location"})
    ENDIF
    DIM people = e{"people"}
    FOR j = 0 TO LEN(people) - 1
        DIM p = people[j]
        DIM cn$ = CHR$(34) + Plain$(p{"name"}) + CHR$(34)
        DIM to$ = "mailto:" + p{"mail"}
        ICAL.ADDPROP(cal, ev, "ATTENDEE", to$, "CN=" + cn$)
    NEXT j
NEXT k
RETURN ICAL.TEXT$(cal)
ENDFUNC

' A name without the characters that end a parameter in an .ics line.
FUNC Plain$(name$)
    DIM out$ = name$
    DIM bad = [CHR$(34), ";", ":", ",", CHR$(13), CHR$(10)]
    DIM k = 0
    FOR k = 0 TO LEN(bad) - 1
        DIM c$ = bad[k]
        out$ = REPLACE$(out$, c$, " ")
    NEXT k
    DO WHILE INSTR(out$, " ") >= 0
        out$ = REPLACE$(out$, " ", " ")
    LOOP
    RETURN TRIM$(out$)
ENDFUNC

' A time as the Date line of a mail, in local time with the offset
' from UTC, such as "Mon, 05 Oct 2026 09:30:00 +0200".
EXPORT FUNC DATE_LINE$(t)
    DIM p = DATE.PARTS(t)
    DIM y = CLNG(p{"year"})
    DIM mo = CLNG(p{"month"})
    DIM d = CLNG(p{"day"})
    DIM h = CLNG(p{"hour"})
    DIM mi = CLNG(p{"minute"})
    DIM s = CLNG(p{"second"})
    DIM wall = DATE.UTC(y, mo, d, h, mi, s)
    DIM offset = CLNG(ROUND((wall - t) / 60))

```

```

    RETURN MAIL.DATETEXT$(y, mo, d, h, mi, s, offset)
ENDFUNC

' A mail from MAILs as the text of an .eml file.
EXPORT FUNC EML$(mail)
    DIM msg = MAIL.MESSAGE(mail{"from"}, mail{"to"}, mail{"subject"})
    IF LEN(mail{"cc"}) > 0 THEN MAIL.CC(msg, mail{"cc"})
    MAIL.PLAIN(msg, mail{"text"})
    MAIL.SETDATE(msg, DATE_LINE$(CDBL(mail{"received"})))
    RETURN MAIL.BUILD$(msg)
ENDFUNC

' The file name of a mail: when it came and its subject, with only
' letters, digits, spaces, dots and dashes, such as
' "2026-10-05 0930 Budget 2027.eml".
EXPORT FUNC FILE_NAME$(mail)
    DIM t = CDBL(mail{"received"})
    DIM day$ = FORMAT_DATE(t, "%Y-%m-%d %H%M")
    DIM subject$ = mail{"subject"}
    DIM keep$ = ""
    DIM k = 0
    FOR k = 0 TO LEN(subject$) - 1
        DIM c$ = MID$(subject$, k, 1)
        DIM ok = (c$ >= "a" AND ALSO c$ <= "z") ORELSE c$ = " "
        IF c$ >= "A" AND ALSO c$ <= "Z" THEN ok = TRUE
        IF c$ >= "0" AND ALSO c$ <= "9" THEN ok = TRUE
        IF c$ = "." ORELSE c$ = "-" THEN ok = TRUE
        IF ok THEN keep$ = keep$ + c$ ELSE keep$ = keep$ + " "
    NEXT k
    DO WHILE INSTR(keep$, " ") >= 0
        keep$ = REPLACE$(keep$, " ", "")
    LOOP
    keep$ = TRIM$(LEFT$(TRIM$(keep$), 60))
    IF keep$ = "" THEN keep$ = "mail"
    RETURN day$ + " " + keep$ + ".eml"
ENDFUNC

```

How it works

1. `CREATEOBJECT("Outlook.Application")` connects to Outlook. If it is already open, the recipe uses it; if not, Outlook starts in the background. The recipe never quits it, so an Outlook you have open stays open.
2. `OLBRIDGE.EVENTS` opens the default calendar and asks it with `Restrict` for the appointments that overlap today. Outlook reads the dates in that filter in your Windows date format, such as 05.10.2026 00:00 in Germany or

10/5/2026 00:00 in the United States; OFFICEKIT.SHORT_DATE\$ finds that format and OFFICEKIT.LOCAL_DATE\$ writes it. IncludeRecurrences makes a weekly meeting show up as today's occurrence.

3. For each appointment it keeps the subject, the times, the place and the people with their names and addresses. A colleague in the company address book has an internal Exchange address; the recipe asks for the normal mail address instead, the one X17 looks for in your mails.
4. OLBRIDGE.ICS\$ writes the appointments with the library ICAL, one ATTENDEE line per person with the name in CN, as Outlook's own export does.
5. OLBRIDGE.MAILS goes through the inbox from the newest mail on and keeps sender, recipients, subject, text and the time it arrived. Meeting requests and read receipts are left out. OLBRIDGE.EML\$ builds an .eml file of each with the library MAIL, and OLBRIDGE.FILE_NAME\$ names it after its time and subject. A mail that is already in the folder is not written again.
6. With --send, OUTBOX.SEND_OUTLOOK reads each .eml file of the outbox and lets OFFICEKIT.OUTLOOK_SEND create a new Outlook mail with the same recipients, subject, text and attachments, and send it. A mail that went out moves to outbox/sent, as with a mail server.

Run it

See what would happen first:

```
jdbasic outlook_bridge.jdb --dry-run
would read 1 day(s) of appointments from 2026-10-05
  into C:/Users/lena/Documents/AutomateWork/calendar.ics
would save the newest 50 mails of the inbox
  into C:/Users/lena/Documents/AutomateWork/saved mail
```

Then without --dry-run:

```
jdbasic outlook_bridge.jdb
4 appointment(s) in C:/Users/lena/Documents/AutomateWork/calendar.ics
50 new mail(s) in C:/Users/lena/Documents/AutomateWork/saved mail
```

Open calendar.ics with a text editor to see what Outlook gave, and run the Meeting Briefing after it. To try sending, put one mail to yourself into the outbox, run jdbasic outlook_bridge.jdb --send, and look into *Sent Items*.

Schedule it

The wizard plans it on weekdays at seven, half an hour before the Meeting Briefing, and leaves it off at first because it needs classic Outlook. Switch it on in the wizard once a run by hand worked.

Make it yours

The settings are the [outlook_bridge] part of work.conf:

```
[outlook_bridge]
calendar_file = "~/Documents/AutomateWork/calendar.ics"
mail_folder = "~/Documents/AutomateWork/saved mail"
days = 1
mails = 50
```

- **The week ahead.** Set days = 7, and the calendar file holds the next seven days, for the Calendar Bridge (X05) or your phone.
- **Calendar only.** Set mails = 0, and the inbox is not read.
- **Other folders.** The inbox and the calendar are Outlook's default folders. GetDefaultFolder in EVENTS and MAILS names them by a number; a folder of your own is found with Folders by its name.

When it goes wrong

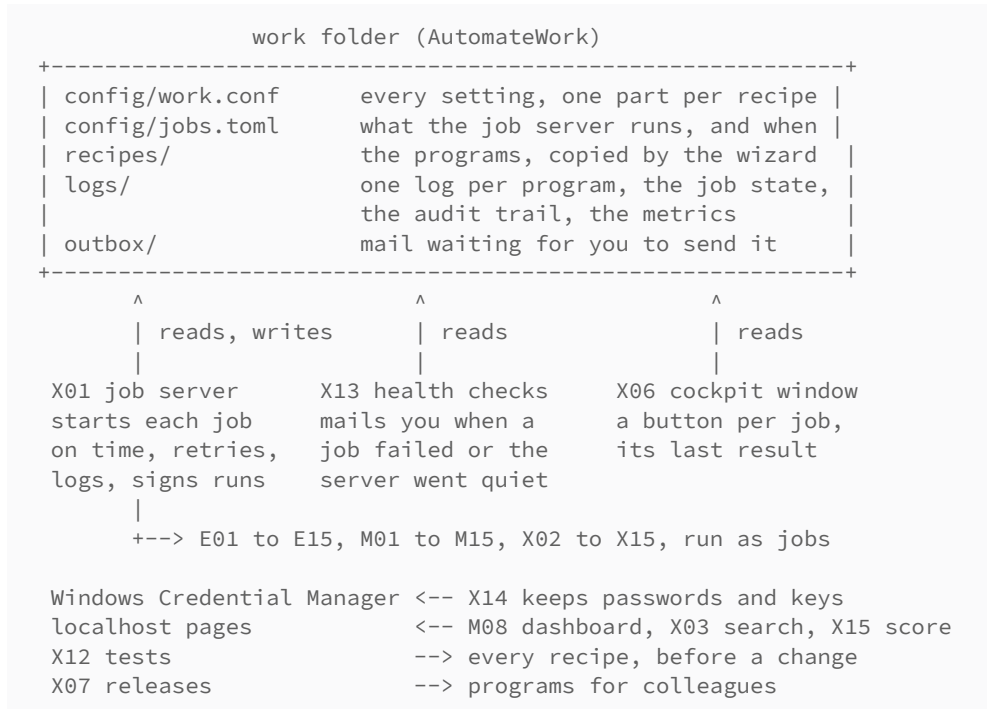
- **"COM: Cannot find ProgID 'Outlook.Application'":** classic Outlook is not installed, or only the new Outlook is. The recipe cannot work with the new one.
- **The run hangs:** Outlook is waiting for an answer, either the security question above or a dialog asking for your password. Look for an Outlook window, answer it, and talk to IT about the warning.
- **"0 appointment(s)" although your day is full:** the appointments are in another calendar than the default one, for example a shared team calendar. See *Other folders* above.
- **"not sent: invoice.eml: ...":** Outlook refused that mail, most often because an address is not valid. The mail stays in the outbox; correct it and send again.

Balance dividend

About 20 minutes a week for anyone whose calendar and mail live in Outlook, and the other recipes of this book in a company where Outlook is the only way out.

The Whole Picture

Lena's team now runs on about twenty programs. None of them is large, and each can be read in an afternoon. What makes them a system is that they share a few files and agree on who writes which one. This page shows those files and the programs around them.



Four rules hold the system together. They are the same habits this chapter started with, now written as who may do what:

1. **One program writes each file.** The job server writes the job state, each recipe writes its own log, the wizard writes `work.conf`, and you write `jobs.toml`. Everything else only reads. Two writers of one file is how a system loses data at three in the morning.
2. **Settings live in `work.conf`, secrets in the Credential Manager.** A setting can be read by anyone who can read the folder, so a password never becomes one.
3. **Mail waits in the outbox.** Programs prepare messages; a person, or a health check with a fixed address of your own, sends them.

4. **Nothing changes without a test.** X12 runs every recipe's test. Run it before you copy a changed recipe into the work folder, and the Monday report keeps arriving.

Where to go from here

Start with the job server and the health checks, then move recipes over to the server one at a time, each after a week of dry runs. Add the cockpit when a colleague asks how to start something, and the release tool of X07 when they ask for a copy. Chapter 6 is about the months after that: what breaks, how to notice, and how to talk to IT about all of it.

Chapter 6: Living With Your Automations

A recipe that works on the day you set it up is the easy part. The harder part comes three months later, when a folder has moved, a password has changed, a colleague wants a copy and nobody remembers why a task runs at half past six. This chapter is about those months.

Mia has run her easy recipes since October. Jonas has the medium ones for his team's documents and mail, and Lena keeps a dozen jobs running under the job server of Chapter 5. Each of them meets the questions of this chapter at a different scale, and the answers are the same for all three:

1. **Look at your automations once a month.** A short check finds most problems before they cost you anything.
2. **Change a program the safe way.** Copy it, write the test, make the change, run it dry. The section on changing a program walks through a real change to the setup wizard.
3. **Share without becoming the help desk.** A recipe you hand out needs a few things it does not need on your own computer.
4. **Talk to IT before they talk to you.** Most IT departments are glad to answer a clear question, and the recipes of this book give you clear answers to theirs.
5. **Retire what no longer pays.** An automation you do not need is a small risk with no return.
6. **Read your ledger.** After three months the ledger of Chapter 1 tells you which recipes give back time, and where that time went.

The chapter ends with the question the whole book is about: what to do with the hours.

Maintenance and the Monthly Check

Programs do not wear out, but the world around them moves. The recipes of this book were written to fail in a visible way: they print what is wrong, they stop before they touch a file they cannot handle, and they leave your data where it was. That is a good start. It only helps if somebody looks.

What breaks

Almost every failure in the first year comes from one of five places.

Paths. A folder is renamed, a network drive gets a new letter, a new laptop comes with a different user name. The recipe looks for its folder, does not find it and stops. The fix is one line in `work.conf`, or a run of the wizard, which finds the settings again and lets you point the recipe at the new place.

Passwords. Your mail password expires every ninety days, and the recipes of Chapter 4 do not know. A `--send` then ends with a message from the mail server, and the messages stay in the outbox, so nothing is lost. If you keep the password in the Windows Credential Manager, update the entry there; recipe X14 shows how.

Web pages. The Page Watcher of E14 and the Web Harvester of X11 read pages that other people build. When a shop redesigns its product page, the value the recipe looked for is no longer where it was. The recipe reports that it found nothing, which is the right answer, and you change the markers of E14 or the selectors of X11 to match the new page.

Files from other programs. An export from the accounting system gets a new column, an Excel template gains a sheet, a calendar program writes its dates in another form. Recipes that read such files check what they expect and say which column or field is missing.

Settings you typed. A quote missing in `work.conf` makes the whole file unreadable, and every recipe stops with a message from the settings reader, such as `CONF.TOML: unterminated string`. The line number in that message belongs to the reader, not to your file, so look at the lines you changed last; a quote that opens and never closes is the usual cause.

Watch out

Most easy and medium recipes print what they do, and the Task Scheduler does not keep that output. When a scheduled recipe fails, the only trace on its own is the task's last result. Run the recipe by hand to see its message, or let the job server of X01 start it, which writes every run into a log.

The monthly check

Pick a fixed day, such as the first Monday of the month, and give the check fifteen minutes. Mia does it with her first coffee; Lena has turned most of it into a job.

1. **Ask the doctor.** It checks the settings file, the path to `jdbasic`, the copied recipes and their tasks:

```
jdbasic wizard\setup_console.jdb --doctor
```

Everything is in order. is the answer you want. Each problem it finds is one line, and running the wizard again repairs most of them.

1. **Look at the last results.** The Task Scheduler remembers when each task last ran and how it ended. In the *AutomateWork* folder the columns *Last Run Time* and *Last Run Result* show it; a result of `0x0` means the program ended without an error. From a command prompt:

```
schtasks /Query /FO LIST /V /TN "AutomateWork\E01 Downloads Butler"
```

A task that has not run for weeks usually means the computer was off or asleep at its time. A result other than `0x0` means the program stopped with an error; run it by hand to read the message.

1. **Run each recipe dry once.** `--dry-run` shows what a recipe would do today without doing it. Read the list: are these the files you expect, the mails, the rows? A few recipes that only read and write their own report, such as the Duplicate Finder (E10) or the Disk Space Report (E11), have no dry run because they change nothing.
1. **Empty the outbox.** Mail that waits in outbox for weeks is mail nobody sent. Send it with `--send`, or delete what is no longer needed.

1. **Write down what you found.** The file `problems.txt` from Chapter 2 gets a line for every failure and its fix. After a few months you will see that the same two or three causes come back, and you can fix them for good.

Here are Mia's first entries, from October to December:

```
2026-10-12  E01 did not run on Friday. Laptop asleep at 18:30.
            Nothing to fix; Monday's run sorted both days.
2026-10-29  E08 template for the travel request lost its umlauts.
            Saved the template as UTF-8 again in Notepad.
2026-11-03  Doctor: "E03: program missing at ...". I had renamed
            the recipes folder. Ran the wizard again.
2026-11-17  E14 found no price. The shop changed the page; new
            start marker id="offer-price">.
2026-12-01  E07 sheet empty for Monday. I forgot to start the
            clock. Not a bug; I start it first thing now.
```

None of the five entries was a bug in a recipe. Two needed a setting, one a file saved again, the others a habit or nothing at all. That is typical, and it is the reason to write them down: after a quarter you know which failures need code and which need a habit.

The expert version

Lena runs her jobs under the job server of X01, so most of the check happens on its own:

- The health checks of X13 run every fifteen minutes. Lena has switched `send_alerts` on, so they send her a mail when a job failed or is late, when the server has gone quiet, or when mail has waited in the outbox for too long. On the first Monday she reads the numbers they keep, not each job's log.
- The test runner of X12 runs every recipe's test before she copies a changed recipe into the work folder, and once a month as a job of its own, so a library update that breaks something shows up on a Monday and not at the end of the quarter.
- `jdbasic secrets_audit.jdb --verify` of X14 checks that the audit trail of the job server is whole.

Her monthly check is the doctor, the health report and the outbox. It takes five minutes.

Updates

New versions of `jdBasic` come out a few times a year. Before you install one, run the tests of the recipes you use with the new version in a folder of its own, or with the test runner of X12. Keep the old version until the new one has run your recipes for a week. The same goes for Windows updates on a work computer: you do not choose when they arrive, but the monthly check after one is worth doing a little more carefully.

Changing a Program: A Monthly Schedule

Sooner or later a recipe does almost what you want. This section walks through one real change from start to end. It is small on purpose: the steps matter more than the code.

Jonas runs the Timesheet for the Boss (M15). His department head wants the sheet on the first of the month, and the wizard knew schedules by the day, the week, the hour and the minute, but none by the month. A weekly run on Mondays worked, because the recipe writes each month only once, but Jonas wanted the schedule to say what it means. So the wizard had to learn a new kind of schedule: `monthly 1 08:00`, the first day of every month at eight.

1. Find the place

The wizard turns a schedule such as `weekly fri 16:00` into the arguments of `schtasks`, the Windows program that creates scheduled tasks. Search the wizard's files for the word `weekly`; Notepad's *Find* does it, or from a command prompt in the book folder:

```
findstr /n "weekly" wizard\wizard.jdb
```

The hits lead to one function, `SCHEDULE_ARGS` in `wizard/wizard.jdb`. It splits the schedule into words and has one line per kind of schedule. A new kind means a new line there. The comment above the function lists the kinds it understands, so it changes too.

Before writing anything, find out what the other side wants. Windows explains its own tools:

```
schtasks /Create /?
```

The help says that a monthly task takes `/SC MONTHLY` and a day of the month with `/D`, from 1 to 31. That raises a question the code has to answer: what about the 31st in a month that has only 30 days? The safe answer is not to allow it. Every month has the days 1 to 28, so the wizard accepts those and refuses the rest with a message that says why.

2. Copy before you change

Copy the folder wizard to wizard_before before you touch it. If anything goes wrong, you delete the folder you broke and rename the copy back. People who know git commit instead; either way, there must be a way back that takes one minute.

3. Write the test first

The wizard has a test, wizard/wizard_test.jdb, that checks every kind of schedule without creating a single task. A change starts there, with the checks that the new schedule has to pass. A helper turns a schedule into the last six schtasks arguments, where the schedule itself sits:

```
FUNC Monthly$(r, when$)
    DIM args = WIZARD.SCHEDULE_ARGS(r, when$, "j", "p", "c")
    RETURN JOIN(TAKE(-6, args), " ")
ENDFUNC
```

```
FUNC LateInMonth(r)
    RETURN Monthly$(r, "monthly 30 08:00")
ENDFUNC
```

and the checks name what the change must do: the first of the month, another day, a day that some months do not have, and the line in the plan the wizard shows on its last page:

```
DIM once$ = Monthly$(e01, "monthly 1 08:00")
TESTKIT.EQ(once$, "/SC MONTHLY /D 1 /ST 08:00", "once a month")
DIM mid$ = Monthly$(e01, "Monthly 15 17:30")
TESTKIT.EQ(mid$, "/SC MONTHLY /D 15 /ST 17:30", "up to the 28th")
TESTKIT.THROWS(LateInMonth@, e01, "a day not in every month")
DIM plan$ = WIZARD.PLAN_LINE$("Sheet", "monthly 1 08:00")
TESTKIT.EQ(plan$, "schedule Sheet: monthly 1 08:00", "in the plan")
```

Run the test before you change the wizard:

```
jdbasic wizard\wizard_test.jdb
```

It stops at the first new check, with the message the old wizard gives for every schedule it does not know. Without the line number it ends with, it reads:

```
Error #99: WIZARD: cannot schedule "monthly 1 08:00" for E01
```

That failure is the point of this step. It proves that the test looks at the right thing; a test that passes before the change proves nothing about the change.

4. Make it pass

Two pieces of code do the work. The first checks the day of the month and answers it as schtasks wants it:

```
' A day of the month from 1 to 28, the days every month has.
FUNC MonthDay$(d$)
    IF LEN(REGEX.FINDALL("[0-9]{1,2}$", d$)) > 0 THEN
        DIM n = CINT(d$)
        IF n >= 1 ANDALSO n <= 28 THEN RETURN STR$(n)
    ENDIF
    THROW "WIZARD: day " + d$ + " is not in every month; use 1 to 28"
ENDFUNC
```

The second is a new block in SCHEDULE_ARGS, next to the lines for the other schedules:

```
IF n = 3 ANDALSO parts[0] = "monthly" THEN
    IF IsClock(parts[2]) THEN
        DIM sc = ["/SC", "MONTHLY", "/D", MonthDay$(parts[1])]
        RETURN APPEND(APPEND(head, sc), ["/ST", parts[2]])
    ENDIF
ENDIF
```

parts holds the words of the schedule, so monthly 1 08:00 gives parts[1] = "1" and parts[2] = "08:00". IsClock was already there for the other schedules and checks that the time looks like a time. A monthly schedule with a time it cannot read leaves the block and reaches the last line of the function, which refuses it like any schedule the wizard does not know. The comment above SCHEDULE_ARGS gains "monthly DAY HH:MM" in its list.

Run the test again. Every check passes, the old ones included, which shows that the new block did not disturb the others:

```
57 passed, 0 failed, 0 skipped
```

5. Run it dry

A passing test says the code does what the test asks. The dry run says the whole wizard still works. Start the wizard, pick the Timesheet for the Boss on the third page, type `monthly 1 08:00` as its schedule, and on the last page tick *Only show what would happen* before you press *Set up now*. The plan shows the new line:

```
schedule Timesheet for the Boss: monthly 1 08:00
```

Nothing has been created yet. When the plan looks right, untick the box and set up for real.

6. Change the default

The schedule a recipe starts with comes from its `recipe.toml`. For M15 that line now reads:

```
schedule = "monthly 1 08:00"
```

The wizard shows it the next time you run it, and every reader of the book gets the new schedule as the default.

What the change does not do

Be honest about the limits of a change, in the code and on paper. The Task Scheduler does not catch up a task whose day passed while the computer was off, and the first of a month falls on a weekend about two months in seven. For a computer that is off at weekends, the old weekly `mon 08:00` is still the better schedule for M15, and its page in Chapter 4 says so. The job server of X01 catches up missed runs, which is one reason Lena moved her monthly jobs there.

The pattern

The same steps work for any change, from a new setting to a new recipe:

1. Find the place, and find out what the other side expects.
2. Make a copy you can go back to.
3. Write the checks first and watch them fail.

4. Change the code until every check passes, the old ones too.
5. Run the program dry and read what it would do.
6. Only then let it act, and change the defaults last.

It looks slow for a change of a dozen lines. It is the reason the change still works next year.

Sharing Without Becoming the Help Desk

Jonas's Friday Status Mail (M06) worked so well that two other team leads asked for it. A week later one of them called because it did not find his notes, and the other wanted the mail in a different order. Sharing a recipe is a good thing to do. It also changes it: on your own computer it is a tool, on five computers it is a small product, and a product needs a few things a tool does not.

Before you hand it out

Run through this list once, on a copy of the recipe:

- **Nothing of yours is in the code.** Folders, names, addresses and times belong in `work.conf`, with a default that makes sense for somebody else. Search the program for your user name and your mail address; neither should appear.
- **The test passes on a fresh folder.** Every recipe's test works in a folder of its own, made with `MKTEMP$`. If it only passes on your computer, it depends on something you forgot.
- **The dry run is the first thing they see.** Tell your colleague to start with `--dry-run` and read the list, exactly as you did.
- **There is a page of text.** What the recipe does, which settings it reads, what it changes, and what to do when it fails. The recipe pages of this book are written in that order; a colleague's copy needs a short version of the same.
- **It has a version.** A date in the first comment is enough, such as version 2026-11-02. When somebody reports a problem, the first question is which version they have.

Jonas's page for the Friday Status Mail fits on one screen:

```
Friday Status Mail (M06), version 2026-11-02
Owner: Jonas Example. Questions: the "automation" channel.

What it does: every Friday at 15:00 it reads the notes in the
status folder, builds the week's status mail and puts it into
your outbox. It sends nothing on its own.

Settings, in the [friday_status] part of work.conf:
notes_folder  where your notes are (~/Documents/Team/status)
to            who gets the mail, a list of addresses
```

```

subject      the subject line; {week} becomes 2026-W45
clear_done   true removes the done lines from your notes

```

What it changes: a new .eml file in the outbox; with
clear_done = true also your notes files.

```

First run: jdbasic friday_status.jdb --dry-run
Send:      jdbasic friday_status.jdb --send

```

When it fails: send the version, the full message and the
output of the dry run to the channel above.

Three ways to share

The recipe folder. Your colleague installs jdBasic, copies the recipe folder into the book's recipes folder, next to the others of its level, and runs the setup wizard, which lists it and writes their own work.conf. This is the right way for people who will change the recipe themselves.

A compiled program. Recipe X07 builds an .exe with its runtime, a version file and checksums, and puts it on a shared drive. Its update command brings a colleague's copy to the newest version and checks every file before it changes anything. Your colleagues need no jdBasic of their own. Building the program needs the release of jdBasic with the native compiler, and handing it out needs the approval of IT, which the next section is about.

The cockpit. The Work Cockpit of X06 is a window with a button for every automation. For colleagues who never open a command prompt, it turns "run this with --dry-run" into a click.

Rules that keep you out of the help desk

1. **One place for questions.** A mailbox, a channel in your chat program or a page on the team drive. Questions in the corridor are answered once and forgotten; questions in one place are answered once and found again.
2. **A report has three parts.** Ask for the version, the full message and the output of the dry run. With those three you can find most problems without visiting anybody's desk.
- 3.

One version for everybody. When a colleague wants the mail in a different order, find out whether it can be a setting. If it can, everybody gets the setting. If it cannot, it is their copy, and they maintain it.

4. **Say what you support.** "I fix bugs in the version on the shared drive, on Thursdays" is a fair promise. "I keep it running for everybody" is a second job.
5. **Find a second owner.** When the whole team depends on a recipe, somebody besides you should be able to read it, run its test and change a setting. Show them once, with this book open at the recipe's page.

When you leave

People change teams, and automations stay behind. Before you go, write down which recipes run where, under whose account, with which settings, and who owns them now. If nobody wants to own a recipe, switch it off and say so; an automation nobody owns is worse than none, because people rely on it until the day it breaks.

Talking to IT and Security

Earlier chapters sent you here more than once: when Windows refuses to start a program from your own folder, when you want to hand a compiled recipe to colleagues, when a recipe talks to a service outside your computer. This section is the conversation itself: what to show, what IT will ask, and answers you can give without stretching the truth.

Most IT departments are not against automation. They are against surprises: a program nobody approved, a password in a text file, a port that is open to the whole network. The recipes of this book were written to avoid exactly those, which makes the conversation shorter than you might fear.

When to ask

- **Before you install jdBasic on a work computer**, if your company allows only software that IT installs. Chapter 2 points here for that case.
- **Before you hand out a program**, compiled or not. A program you compiled yourself is not from a known publisher, and many companies allow only those.
- **Before a recipe talks to a company system** such as the ticket system of X04 or the calendar of X05, which needs an app registration or a token that IT issues.
- **Before a recipe sends company data to an outside service**, such as the language model of X02 and X17.
- **Before a recipe types or clicks for you**, as E16, M16 and X16 do. Some companies forbid programs that send key strokes.

What to show

Bring a one-page description and the folder. The description answers the questions below in your own words; the folder lets IT check them.

The program. jdBasic is one program, `jdbasic.exe`, with a few libraries next to it. It needs no administrator rights and installs no service. The recipes write into the folders named here and, while they run, into the Windows folder for temporary files. The releases come from the project page on GitHub; the *Digital*

Signatures tab in the file's properties shows who signed a release. *jdBasic* makes no network connection of its own; only a program that asks for one does.

Where things live.

What	Where
Settings	Documents\AutomateWork\config\work.conf
The recipes you use	Documents\AutomateWork\recipes
Shared modules	.jdbasic\lib in your user folder
Logs and reports	inside Documents\AutomateWork
Scheduled tasks	the folder <i>AutomateWork</i> of the Task Scheduler

The scheduled tasks run under your own account, only while you are logged in, with your rights and no more.

What each recipe does. Every recipe is a text file IT can read, with a *recipe.toml* that names its program, its modules, its settings and its schedule. Each one also has a test, and each one that changes files has a dry run that shows what it would do.

What talks to the network. Most recipes touch only your own files. These are the ones that reach further:

Recipe	Talks to	When
E14 Page Watcher	the web pages you name	on its schedule
M01, M03, M06, M15, M18, X02, X09, X12, X17	your mail server	only with --send
X19 Outlook Bridge	your mail server, through Outlook	only with --send
X02 Inbox Assistant	a language model service, or one on your computer	when you run it
X17 Meeting Briefing	the same, unless <code>use_model = false</code>	on its schedule
X04 Ticket Sync	the ticket system	when you run it
X05 Calendar Bridge	Microsoft 365	on its schedule
X11 Web Harvester	the web pages you name	on its schedule

Recipe	Talks to	When
X07 Tools for Colleagues	the release folder or an address inside the company	on check and update
X13 Health Checks	your mail server	only with <code>send_alerts</code> on

What listens. The Personal Dashboard (M08), the search page of Find Anything (X03), the Balance Score (X15) and the Spreadsheet to Web App (X18) show a page in your browser. They listen on localhost only, on a port set in `work.conf`; no other computer can open them. The AI tools of X10 run over a pipe between the agent and `jdBasic`, not over the network, and they start with `--tools-only`, so the agent sees only the tools you wrote and none of `jdBasic`'s own.

What drives other programs. A few recipes work through programs you already have instead of through files. E18 Office to PDF, M18 Excel Refresh and X19 Outlook Bridge start Word, Excel or classic Outlook through COM, the interface Office offers other programs for this purpose. They do what you could do by hand, under your account and with your rights. E16 Form Filler, M16 Macro Player and X16 Desktop Robot type and click into windows through the same Windows input a keyboard uses. They stop as soon as the mouse is pushed into the top-left corner of the screen, and they belong on a screen that nobody else can see or use. Outlook may ask before a program sends mail through it; whether that question stays is for IT to decide.

Passwords and keys. No recipe writes a password into `work.conf` or any other file of the work folder.

- Mail goes into an outbox first. Sending asks for the password, which the prompt does not show, or reads it from the Windows Credential Manager if you stored it there yourself under `AutomateWork/mail`.
- The Credential Manager keeps entries encrypted for your Windows account. Recipe X14 stores and lists them, and every entry of the book starts with `AutomateWork/`, so IT can see them all in one place.
- Keys for outside services come from environment variables, such as `OPENAI_API_KEY` for X02 or `TICKETS_TOKEN` for X04, or from a prompt.

- The Calendar Bridge (X05) keeps its refresh token in `.automatework\calendar_token.json` in your user folder, outside the work folder, because it has to survive a restart. That file is as sensitive as a password, and its page says so.

Questions IT will ask

"Who maintains this when you are away?" You, and the second owner the previous section asked for. Show the page of text that comes with each recipe and the monthly check of this chapter.

"Can we see the code?" Yes, all of it. Every recipe is a plain text file, and this book prints and explains each one.

"Is the program you hand out signed?" A program you compiled with X07 is not signed unless IT signs it. Some companies sign approved internal tools with their own certificate; ask whether yours does. The checksums that X07 writes show that a copy was not changed on the way, but they do not replace a signature.

"Will our antivirus complain?" It may. A new program from an unknown publisher is exactly what such software looks at more closely. Ask IT for the approved way: a signed build, an approved folder, or an exception for the release folder on the shared drive. Do not look for a way around it.

"What data leaves the computer?" Only what the table above lists, and only for the recipes you switched on. For X02 and X17 the text of your mail goes to the language model service you configured, unless you run a model on your own computer; that is a question for your data protection rules before you start, not after.

"What if it goes wrong?" The recipes do not overwrite files, show their plan with `--dry-run` before they act, and stop with a message instead of guessing. Mail waits in the outbox until a person sends it. The worst case for most recipes is a chore that did not happen.

"Why not use the tools we already have?" A fair question. If IT offers a tool that does the same job, such as rules in the mail program or a flow in a platform the company already pays for, compare them honestly. The recipes are small and readable; a supported tool has a team behind it. Sometimes the answer is to use the recipe as the precise description of what the supported tool should do.

A first mail

A short mail is often all it takes to start. Lena sent this one before she set up the job server:

Subject: Request: small automation scripts on my work PC

I would like to run a few small scripts that tidy my folders and prepare reports. They use `jdBasic`, a single program without installer or admin rights, from github.com/AtomijD/jdBasic. Everything runs under my account and stays in `Documents\AutomateWork`. No passwords are stored in files. Two scripts show a page in my browser on localhost only. One sends mail through our server when I start it by hand.

I can show you the folder and the code at any time. Is this fine, or is there a process I should follow?

If the answer is no

Then the answer is no, and you run nothing on that computer. The work of this book is not lost: the audit of Chapter 1, the ledger and the recipe pages describe precisely which chores cost you time and what a tool would have to do. That is a good basis for a request for a tool IT does support, and for the next conversation in half a year.

When an Automation Should Be Retired

Every recipe in this book was worth its time when somebody wrote it. Not every one stays worth it. Work changes, a chore disappears, a better tool arrives. An automation that no longer pays still costs a little: it runs, it can fail, it reads files and sometimes sends mail, and somebody has to keep it in the monthly check.

Signs that it is time

- **The ledger shows almost nothing.** Mia's Printable Week (E13) gave her 25 minutes in three months, because she plans her week in a calendar now. The recipe still worked; it was no longer needed.
- **You check its work by hand anyway.** If you open every file it sorted to see whether it sorted right, the recipe saves you nothing and adds a step. Either fix what makes you doubt it, or retire it.
- **It needs fixing more often than it saves.** A Page Watcher whose page changes every month costs more time in repairs than it gives back.
- **Nobody reads what it makes.** A report that goes to five people and is opened by none is a chore for the computer instead of for you, and it still fills five inboxes.
- **The chore moved.** The team changed its ticket system, the reports go to a new platform, the colleague who needed the sheet left.

How to retire it

1. **Tell the people who use its results.** If a recipe sends a mail or writes a report that others read, they should hear it from you before it stops.
2. **Switch it off in the wizard.** Untick *Use this recipe* on the third page and set up again. The wizard removes the recipe's tasks from the Task Scheduler and says so for each one. For a job under the server of X01, delete its block from `jobs.toml`.
3. **Keep the folder for a month.** Leave the recipe and its part of `work.conf` where they are. If you miss it, switching it on again costs one click.
4. **Then clean up.** Delete the recipe's folder from `AutomateWork\recipes`, its part of `work.conf`, and any secrets it kept in the Credential Manager, which X14 removes with `jdbasic secrets_audit.jdb --delete` and the secret's name.

5. **Write one line in the ledger.** The date and the reason. The next time you are tempted to write a similar recipe, that line saves you an evening.

Retiring is not a failure of the recipe or of you. It means the recipe did its job long enough for the work around it to change, and that you noticed.

The Ledger After Three Months

Chapter 1 asked you to keep a ledger: one line a week, with the minutes the recipes gave back and what the time went to. After three months it has about thirteen weeks of lines, enough to answer the questions that matter. This section reads Mia's ledger as an example.

Adding up by recipe

Mia added up her minutes per recipe for the thirteen weeks from October to December. A few lines of `jdBasic` give the share of each recipe in the total:

```
DIM recipes = ["E01", "E03", "E07", "E08", "E13"]
DIM minutes = [390, 160, 420, 300, 25]
DIM total = SUM(minutes)
PRINT "Minutes in thirteen weeks: "; total
PRINT "Hours: "; ROUND(total / 60, 1)
DIM k = 0
FOR k = 0 TO LEN(recipes) - 1
    DIM share = ROUND(100 * minutes[k] / total)
    PRINT recipes[k]; ": "; minutes[k]; " minutes, "; share; "%"
NEXT k
```

```
Minutes in thirteen weeks: 1295
Hours: 21.6
E01: 390 minutes, 30%
E03: 160 minutes, 12%
E07: 420 minutes, 32%
E08: 300 minutes, 23%
E13: 25 minutes, 2%
```

Almost 22 hours in a quarter is more than half a working week. Two recipes, the Time Tracker (E07) and the Downloads Butler (E01), give almost two thirds of it. The Printable Week (E13) gives almost nothing, and the previous section retired it.

Four questions for the numbers

Were the estimates right? Every recipe states its minutes per week in its Balance dividend box. Mia's Downloads Butler gave her 30 minutes a week against the 15 of the box, because her downloads folder was worse than the typical one. The Time Tracker gave her a little more than its box says. The difference

matters less than the habit: a number you measured is worth more than one you were promised.

Which recipe would you miss? Switch one off for two weeks and see. Mia did it with the Mail Templates (E08) and switched them back on after three days.

Where did the time go? This is the column that tells the real story. In Mia's first weeks it said *nothing planned, it disappeared*. From the third week on it named things: leaving on time on two days, an hour on Thursday morning for a project that had waited since spring. Time that had a name in advance stayed hers; time without one was taken by the next request.

What is still on the audit? The chores of Chapter 1 that no recipe covered are still there. Some of them now look like candidates for the template of Chapter 4, because three months of recipes made the pattern visible.

Other ledgers

Jonas keeps his ledger for the team, not for himself: the invoices, the merged reports and the Friday status save the team about two hours a week, and he writes down what the team did with them. His most useful line was the one that showed the status mail saving him most of an hour every Friday afternoon, which he now spends on the one-to-one talks he used to postpone.

Lena lets the Balance Score of X15 do the adding up. It reads her calendar, her time log and the mail in her outbox, and turns the week into one number with a trend. She still writes the last column by hand, because no program knows what the time was for.

Keep going, or stop

If the ledger shows a steady number and a clear last column, keep it for another quarter and then decide whether a line a month is enough. If it shows that the time disappears every week, the ledger has done its job too: the next step is not another recipe, but a decision about what the time is for, written down before the week starts.

What to Do With the Time

This book has counted minutes for more than five hundred pages. It is fair to end with the part no program can do.

Saved time does not stay saved on its own. A free half hour at four in the afternoon is filled by the next mail within minutes, unless it already belongs to something. Every reader whose ledger looked good after three months had one thing in common: they decided what the time was for before it arrived. Mia leaves on time on Wednesdays and Fridays. Jonas holds the one-to-one talks he used to cancel. Lena took back her Thursday mornings for the analysis work that brought her into controlling in the first place.

None of that is impressive, and none of it needs to be. The point of automating a chore is not to fill the hour with more work at a higher speed. It is to have the hour, and to choose.

Some honest limits belong here too. A recipe will not fix a job that asks too much of one person, a team that is too small, or a manager who measures presence instead of results. If your audit showed that most of your week is meetings you cannot decline, the recipes give you an hour or two, and the rest is a conversation with your manager that no program can hold for you. The ledger is useful there as well: it shows in plain numbers where the week goes.

If this book did its job, three things are true now. A handful of programs do the chores you used to repeat. You can read them, change them and switch them off. And there is a line in your ledger, every week, that says what you did with the time. Keep the line. The programs are only there to make room for it.

Appendix A: jdBasic Quick Reference

This appendix lists the part of jdBasic that the programs in this book use, and nothing else. Each entry gives the call with its arguments in the order jdBasic expects them, what it answers, and the trap that catches people most often. The full reference with every builtin is `doc/languages.md` in the jdBasic folder; `jdbasic --help` and the `HELP` command of the interactive mode show it too.

Three rules hold everywhere:

- **Positions count from 0.** The first character of a text, the first element of an array and the result of `INSTR` all start at 0. `IOTA` is the one exception: it counts from 1 unless you ask otherwise.
- **A name that ends in `$` holds text.** `name$` is always a string; functions whose name ends in `$` answer one.
- **Names are not case sensitive.** `Total`, `TOTAL` and `total` are the same variable.

Programs and modules

Statement	What it does
<code>IMPORT CONF, DT</code>	Loads modules; their functions are called as <code>CONF.GET(...)</code>
<code>EXPORT MODULE NAME</code>	First line of a module file such as <code>name.jdb</code>
<code>EXPORT FUNC NAME(a, b)</code>	A function other programs may call
<code>FUNC Name(a, b = 0) ... ENDFUNC</code>	A function; <code>b</code> has a default value

Statement	What it does
SUB Name(a) ... ENDSUB	A procedure that answers nothing
RETURN value	Leaves a function with its answer
END	Ends the program at once
CONST LIMIT = 3	A name whose value cannot change
Name@	The function itself as a value, to hand to SELECT or RETRY.RUN
LAMBDA x -> x + 1	A small function written in place
ASYNC FUNC Worker(...)	A function a worker pool runs in parallel (X08)
ON handler\$ CALL Name	Connects a window event to a function (X06)
DECLARE FUNC ... LIB "x.dll"	Calls a function of a Windows DLL (VAULT in X14)

A program finds its modules in its own folder, then in the folders of the environment variable `JDBASIC_PATH`. The wizard copies the shared modules of the book into such a folder for you.

Variables and values

Form	Meaning
<code>DIM n = 0</code>	A variable with its first value
<code>DIM name\$ = ""</code>	A text variable
<code>DIM items = []</code>	An empty array
<code>DIM m AS MAP</code>	An empty map
<code>DIM m = {"a": 1, "b": "two"}</code>	A map with keys and values
<code>TRUE, FALSE</code>	Truth values
<code>NONE</code>	No value; what a missing map key answers
<code>TYPEOF(x)</code>	"INT64", "FLOAT64", "STRING", "ARRAY", "OBJECT", "DATE", "BOOLEAN" or "NONE"

To find out whether a map has a key, ask `MAP.EXISTS(m, "key")` or `typeof(m{"key"}) = "NONE"`. A test such as `m{"key"} = NONE` does not work, and `"" + m{"missing"}` gives the text `NONE`.

Operators

Operator	Meaning
<code>+ - * /</code>	Arithmetic; <code>+</code> also joins texts
<code>\</code>	Whole-number division: <code>7 \ 2</code> is 3
<code>MOD</code>	Remainder: <code>7 MOD 3</code> is 1
<code>^</code>	Power: <code>2 ^ 3</code> is 8
<code>= < > <= >=</code>	Comparisons; <code>=</code> compares texts too
<code>AND OR NOT</code>	Logic; both sides are always worked out
<code>ANDALSO ORELSE</code>	Logic that stops early: the right side only runs when it matters
<code>x IN items</code>	Whether an array holds a value

Use `ANDALSO` when the right side would fail without the left one, as in `IF LEN(a) > 0 ANDALSO a[0] = "x" THEN`.

Decisions and loops

```

IF n > 10 THEN
    PRINT "many"
ELSEIF n > 0 THEN
    PRINT "some"
ELSE
    PRINT "none"
ENDIF
IF done THEN PRINT "ok" ELSE PRINT "not yet"

FOR k = 0 TO LEN(items) - 1
    PRINT items[k]
NEXT k
FOR k = 10 TO 0 STEP -2
NEXT k
FOR EACH name$ IN names

```

```

NEXT
FOR EACH key$, value IN settings
NEXT

DO WHILE NOT finished
    IF tries > 3 THEN EXITDO
LOOP
DO
    n = n + 1
LOOP WHILE FILE.EXISTS(candidate$)

```

EXITDO leaves a DO loop, EXITFOR a FOR loop. A DIM inside a loop gives the variable a new value on every pass.

Text

Call	Answers
LEN(s\$)	The number of characters
MID\$(s\$, start, n)	n characters from position start, counted from 0; without n to the end
LEFT\$(s\$, n), RIGHT\$(s\$, n)	The first or last n characters
INSTR(s\$, find\$)	The position of find\$, from 0, or -1 when it is not there
INSTR(start, s\$, find\$)	The same, searching from start
REPLACE\$(s\$, old\$, new\$)	Every old\$ replaced
TRIM\$(s\$)	Without spaces at both ends
LCASE\$(s\$), UCASE\$(s\$)	In small or capital letters
SPLIT(s\$, sep\$)	An array of the parts
JOIN(items, sep\$)	The parts joined into one text
STARTSWITH(s\$, p\$), ENDSWITH(s\$, p\$)	Whether the text begins or ends so
STR\$(n), VAL(s\$)	A number as text, a text as number
CINT(x)	A whole number; text that is not a number is an error
CHR\$(n), ASC(c\$)	A character from its code, a code from its character

Call	Answers
REPEAT\$(s\$, n)	The text n times
FORMAT\$(spec\$, v, ...)	Values put into {} holes: {:.2f}, {:>8}
REGEX.MATCH(pattern\$, s\$)	Whether the whole text matches
REGEX.FINDALL(pattern\$, s\$)	Every match
REGEX.REPLACE(pattern\$, s\$, new\$)	Every match replaced; \$1 in new\$ is the first group
-s\$	The characters as an array

Two traps: INSTR answers -1, not 0, when it finds nothing, so test INSTR(...) >= 0; and the start position of INSTR comes first. In FORMAT\$ a width pads with spaces only, so { :05d} does not give leading zeros.

```
DIM s$ = "Invoice 0413.pdf"
PRINT MID$(s$, 0, 7)
PRINT INSTR(s$, "0413")
PRINT INSTR(s$, "x")
PRINT FORMAT$("{:.2f} EUR", 209.8)
PRINT JOIN(SPLIT("a;b;c", ";"), " + ")
```

```
Invoice
8
-1
209.80 EUR
a + b + c
```

Numbers

Call	Answers
INT(x)	The whole part: INT(3.7) is 3
ROUND(x, digits)	Rounded: ROUND(2.345, 2) is 2.35
FLOOR(x), ABS(x)	Rounded down, without sign
SUM(items)	The sum of an array
MIN(items), MAX(items)	The smallest and largest element of an array

MIN and MAX take one array: write `MIN([a, b])` for the smaller of two numbers.

Arrays

Call	Answers
<code>items[0]</code>	The first element
<code>LEN(items)</code>	The number of elements
<code>PUSH(items, v)</code>	Adds <code>v</code> at the end of the array itself
<code>APPEND(items, v)</code>	A new array with <code>v</code> added; the old one stays
<code>SORT(items)</code>	A sorted copy
<code>GRADE(items)</code>	The positions that would sort the array
<code>TAKE(n, items), DROP(n, items)</code>	The first <code>n</code> , or all but the first <code>n</code> ; negative <code>n</code> counts from the end
<code>UNIQUE(items), REVERSE(items)</code>	Without repeats, back to front
<code>SELECT(Fn@, items)</code>	<code>Fn</code> applied to every element
<code>FILTER(Fn@, items)</code>	The elements for which <code>Fn</code> is true
<code>ZEROS(n)</code>	<code>n</code> zeros
<code>IOTA(n)</code>	1 to <code>n</code> ; <code>IOTA(n, 0)</code> gives 0 to <code>n - 1</code>

`APPEND` is a function: write `items = APPEND(items, v)`. `PUSH` changes the array in place and needs no assignment.

```
DIM a = [30, 10, 20]
PUSH(a, 5)
PRINT SORT(a)
PRINT TAKE(2, a)
PRINT SELECT(LAMBDA x -> x * 2, a)
PRINT 10 IN a
```

```
[5, 10, 20, 30]
[30, 10]
[60, 20, 40, 10]
TRUE
```


Maps

Call	Answers
<code>m{"key"}</code>	The value of a key, <code>NONE</code> when it is missing
<code>m{"key"} = v</code>	Sets a value
<code>MAP.EXISTS(m, "key")</code>	Whether the key is there
<code>MAP.KEYS(m)</code>	The keys, in the order they were added
<code>MAP.DELETE(m, "key")</code>	Removes a key

Dates and times

Call	Answers
<code>NOW()</code>	The date and time of this moment
<code>CDATE(text\$)</code>	A date from "2026-10-05", "05.10.2026" or "2026-10-05 08:30:00"
<code>FORMAT_DATE(d, fmt\$)</code>	Text with %Y %m %d %H %M %S %A %B filled in
<code>DATEADD(unit\$, n, d)</code>	d moved by n units: "Y" "M" "W" "D" "H" "N" "S"
<code>DATEDIFF(unit\$, a, b)</code>	From a to b in "D" "H" "N" or "S"
<code>YEAR(d), MONTH(d), DAY(d)</code>	Parts of a date
<code>HOURL(d), MINUTE(d)</code>	Parts of the time
<code>WEEKDAY(d)</code>	0 for Sunday to 6 for Saturday
<code>TICK()</code>	Milliseconds, for measuring how long something takes

In `DATEADD` the number comes before the date. "N" is minutes, because "M" is months. `DATEDIFF` with "D" answers fractions of a day; wrap it in `INT` for whole days. Days and weeks follow the calendar, so adding a day across a change of the clocks keeps the time of day. The `DT` library of Appendix B does the same with seconds you can count with, and knows time zones.

```
DIM d = CDATE("2026-10-05 08:30:00")
PRINT FORMAT_DATE(d, "%A, %d %B %Y")
PRINT FORMAT_DATE(DATEADD("D", 3, d), "%Y-%m-%d")
PRINT DATEDIFF("H", CDATE("2026-10-05"), d)
```

```
Monday, 05 October 2026
2026-10-08
8.5
```

Files and folders

Call	Answers
PATH.JOIN\$(a\$, b\$, ...)	A path from its parts, with the right separator
PATH.DIRNAME\$(p\$), PATH.BASENAME\$(p\$)	The folder, the file name
PATH.EXT\$(p\$)	The extension with its dot: ".pdf"
PATH.NORMALIZE\$(p\$)	The path tidied up
FILE.EXISTS(p\$), FILE.ISDIR(p\$)	Whether there is a file, or a folder
FILE.STAT(p\$)	A map with size, mtime, is_dir, hidden, readonly, exists
FILE.SIZE(p\$)	The size in bytes
FILE.MOVE(from\$, to\$, overwrite)	Moves or renames; into a folder when to\$ is one
FILE.COPY(from\$, to\$, overwrite)	Copies; overwrite is FALSE when left out
DIR\$(pattern\$)	The names in a folder that match, such as "C:/Scans/*.pdf"
MKDIR path\$	Creates a folder and every folder above it that is missing
RMDIR path\$	Removes an empty folder
KILL path\$	Deletes a file
TXTRADER\$(path\$)	A text file as one string
TXTWRITER path\$, text\$	Writes a text file, replacing what was there

Call	Answers
<code>BINREADER\$(path\$), BINWRITER path\$, data\$</code>	The same for bytes
<code>CSVREADER(path\$, sep\$, header, types)</code>	A CSV file as rows of cells
<code>MKTEMP\$(prefix\$)</code>	A fresh path in the temporary folder, not yet created
<code>CD()</code>	The current folder

`DIR$` answers names only; join them with the folder before you open them. `FILE.MOVE` and `FILE.COPY` refuse to overwrite unless you pass `TRUE`. `MKTEMP$` gives a name: create the folder with `MKDIR`.

Other programs and the system

Call	Answers
<code>OS.EXEC(program\$, args)</code>	Runs a program; a map with "EXIT_CODE" and "OUTPUT"
<code>OS.ARGS()</code>	The words on the command line
<code>GETENV\$(name\$)</code>	An environment variable, "" when it is not set
<code>SETENV name\$, value\$</code>	Sets one for this program and the programs it starts
<code>SLEEP ms</code>	Waits so many milliseconds
<code>OS.GETOS()</code>	"WINDOWS" on Windows
<code>OS.SCREENSHOT(path\$, mode\$, title\$)</code>	A picture of a window (X06)

`OS.EXEC` starts the program directly, without a command prompt in between, and hands each element of `args` over as one argument. For a command of `cmd.exe` such as `rmdir`, call `OS.EXEC("cmd", ["/c", "rmdir", folder$])`.

The console

Statement	What it does
PRINT a; b	Prints values next to each other
PRINT a, b	Prints them with a space between
PRINT a;	No new line at the end
INPUT "Name: "; name\$	Reads a line; the typed text shows on screen
WAITKEY\$()	Waits for one key and answers it
OUTPUT.CAPTURE_BEGIN	Collects what PRINT writes from here on
OUTPUT.CAPTURE_END\$()	Stops collecting and answers the text

Never ask for a password with INPUT: it shows on screen. The VAULT module of Appendix B has a prompt that prints stars.

Data, hashes and the web

Call	Answers
JSON.STRINGIFY\$(v)	A map or array as JSON text
JSON.PARSE\$(text\$)	JSON text read back as a map or array
CODEC.SHA256\$(data\$)	The SHA-256 hash as 64 hex digits
HTTP.REQUEST(method\$, url\$, body\$, type\$)	A map with status, body and headers
HTTP.SETTIMEOUT(seconds)	How long a request may take
HTTP.STATUSCODE()	The status of the last request

Despite its \$, JSON.PARSE\$ answers a map or an array. The REQ and JDWEB libraries of Appendix B are the comfortable way to the web.

Errors

```
TRY
    DIM data = XLSX.READ(path$)
CATCH
    PRINT "Could not read "; path$; ": "; ERRMSG$
ENDTRY

IF days < 0 THEN THROW "days must not be negative"
```

ERRMSG\$ holds the message of the error inside CATCH. THROW raises an error with your own text; a TRY further up catches it, and without one the program stops and prints it.

Names you cannot use

Some words are part of the language and cannot name a variable. These fifteen came up while the book was written; DIM stop = 1 stops with "expected variable":

CLS, END, FOR, INPUT, LINE, LOOP, NEXT, ON, PRINT, STEP, STOP, TEXT, THEN, TYPE, USE

The names of builtins are allowed for variables, but not for your own functions: FUNC Count(x) stops with "FUNC COUNT collides with the builtin function COUNT". Names that caught the recipes are COUNT, DAY, JOIN, LEFT, LEN, MAX, MIN, MONTH, NOW, PLACE, SORT, SPLIT, SUM, TICK, TRIM and VAL. A word in front, as in CountFiles or RunStep, avoids all of them.

Appendix B: Library Reference

The libraries are `jdBasic` programs of their own, in the `lib` folder of `jdBasic`, and the book's own modules are in `recipes/lib`. A program loads them with `IMPORT` and calls their functions with the library's name in front: `CONF.GET(cfg, "mail.server", "")`. This appendix lists the functions the recipes call. Each library file starts with a comment that describes the rest; open it in any editor.

Arguments with a value, such as `fallback = 0`, may be left out.

The libraries of Chapter 2

CONF: settings files

Function	What it does
<code>CONF.TOML(path\$)</code>	Reads a TOML file into a map of maps
<code>CONF.TOML_TEXT(text\$)</code>	The same from a text
<code>CONF.INI(path\$, policy\$ = "last")</code>	Reads an INI file; <code>policy\$</code> decides about a repeated key
<code>CONF.ENV(path\$, apply = FALSE)</code>	Reads a <code>.env</code> file; with <code>apply</code> into the environment too
<code>CONF.GET(cfg, path\$, fallback = 0)</code>	A value by a dotted path such as <code>"mail.server"</code> , or the <code>fallback</code>

WORKCONF: the book's settings

Function	What it does
<code>WORKCONF.CONFIG_PATH\$()</code>	Where <code>work.conf</code> is: <code>--config</code> , then <code>AUTOMATEWORK_CONF</code> , then the work folder
<code>WORKCONF.SETTINGS(path\$)</code>	The settings; a missing file is an error that says what to do
<code>WORKCONF.VALUE(cfg, key\$, fallback)</code>	A value by its dotted key
<code>WORKCONF.FOLDER\$(cfg, key\$, fallback\$)</code>	A folder setting with <code>~</code> turned into the home folder
<code>WORKCONF.FLAG(name\$)</code>	Whether <code>--name</code> is on the command line
<code>WORKCONF.ARG\$(name\$)</code>	The word after <code>--name</code> , or ""
<code>WORKCONF.EXPAND\$(path\$)</code>	A path with <code>~</code> expanded
<code>WORKCONF.HOME\$()</code>	The home folder

LOGGER: a diary for programs

Function	What it does
<code>LOGGER.NEW(name\$)</code>	A new logger
<code>LOGGER.LEVEL(lg, level\$)</code>	Drops what lies below "DEBUG", "INFO", "WARN" or "ERROR"
<code>LOGGER.TO_CONSOLE(lg, colour = TRUE)</code>	Writes to the console
<code>LOGGER.TO_FILE(lg, path\$, max_bytes = 0, keep = 3)</code>	Writes to a file, starting a new one at <code>max_bytes</code>
<code>LOGGER.TO_JSONL(lg, path\$, max_bytes = 0, keep = 3)</code>	Writes one JSON object per line
<code>LOGGER.DEBUG(lg, message\$, fields = 0)</code>	A line of the level DEBUG, with a map of fields
<code>LOGGER.INFO(lg, message\$, fields = 0)</code>	The same for INFO
<code>LOGGER.WARN(lg, message\$, fields = 0)</code>	The same for WARN
<code>LOGGER.ERROR(lg, message\$, fields = 0)</code>	The same for ERROR

`LOGGER.TO_FILE` does not create the folder of the file; `MKDIR` it first.

DT: dates as numbers

DT counts time in seconds since 1 January 1970, UTC. `offset` is the offset from UTC in hours that the wall clock uses; `DT.LOCAL_OFFSET()` gives the computer's own.

Function	What it does
<code>DT.NOW()</code>	This moment
<code>DT.PARSE(text\$, offset = 0)</code>	ISO dates and times, RFC 2822, <code>day.month.year</code> ; <code>NONE</code> when it cannot read it
<code>DT.MAKE(yy, mm, dd, hh = 0, mi = 0, ss = 0, offset = 0)</code>	An instant from its parts
<code>DT.PARTS(t, offset = 0)</code>	The parts as a map: year, month, day, hour and more
<code>DT.FMT\$(t, fmt\$, offset = 0)</code>	Text with <code>%Y %m %d %H %M %S %A %B</code> and more
<code>DT.ISO\$(t, offset = 0)</code>	2026-09-12T15:00:00+02:00
<code>DT.WEEKDAY\$(t, lang\$ = "en", offset = 0)</code>	The name of the day
<code>DT.ADD(t, spec\$, offset = 0)</code>	Moved by "+1 month -2 days" or "2w"
<code>DT.DIFF(a, b, unit\$, offset = 0)</code>	From a to b in seconds, minutes, hours, days, weeks, months or years
<code>DT.STARTOF(t, unit\$, offset = 0)</code>	The start of its day, week, month, quarter or year
<code>DT.ENDOF(t, unit\$, offset = 0)</code>	The last second of it
<code>DT.ISOWEEK(t, offset = 0)</code>	A map with "year" and "week"
<code>DT.IS_LEAP(yy)</code> , <code>DT.DAYS_IN(yy, mm)</code>	Leap years and the length of a month
<code>DT.LOCAL_OFFSET()</code>	The computer's offset from UTC in hours

DF: tables

Function	What it does
<code>DF.READCSV(path\$, delim\$ = ",", types = 0)</code>	A CSV file with a header line as a table
<code>DF.FROM(matrix, names)</code>	A table from rows and column names

Function	What it does
<code>DF.FROMROWS(rows)</code>	A table from an array of maps
<code>DF.NAMES(df), DF.NROWS(df)</code>	The column names, the number of rows
<code>DF.COL(df, name\$)</code>	One column as an array
<code>DF.ROW(df, i)</code>	One row as a map
<code>DF.WHERE(df, mask)</code>	The rows where the mask is true
<code>DF.SORT(df, name\$, desc = FALSE)</code>	Sorted by a column
<code>DF.GROUPBY(df, keys, aggs)</code>	One row per key, with sums, counts or means
<code>DF.WRITECSV(df, path\$, delim\$ = ",")</code>	Writes a CSV file
<code>DF.TEXT\$(df, n_rows = 10, opts = 0)</code>	The first rows as a text table

XLSX: Excel files

Function	What it does
<code>XLSX.NEW()</code>	A new workbook
<code>XLSX.SHEET(wb, name\$, rows, header = TRUE)</code>	A sheet from rows; the first row is bold
<code>XLSX.COLUMN(s, col, opts)</code>	Width and number format of a column, by letter or from 1
<code>XLSX.STYLE(s, ref\$, opts)</code>	Fill, bold and format of a cell or a range
<code>XLSX.FREEZE(s, rows, cols = 0)</code>	Keeps the first rows and columns in view
<code>XLSX.WRITE(path\$, wb)</code>	Writes the file
<code>XLSX.READ(path\$)</code>	Every sheet as rows: a map from sheet name to rows
<code>XLSX.SHEETS(path\$)</code>	The names of the sheets

`XLSX.WRITE` takes the path first, `DOCX.WRITE` the document first. Dates come out of `XLSX.READ` as the serial number Excel stores.

DOCX: Word documents

Function	What it does
<code>DOCX.NEW()</code>	A new document
<code>DOCX.TITLE(doc, text\$)</code>	The title in the document properties
<code>DOCX.LANGUAGE(doc, tag\$)</code>	The language of the spelling check, such as "en-US"
<code>DOCX.HEADING(doc, text\$, level = 1)</code>	A heading of level 1 to 3
<code>DOCX.PARAGRAPH(doc, text\$, align\$ = "")</code>	A paragraph; bold and <i>italic</i> are kept
<code>DOCX.BULLETS(doc, items), DOCX.NUMBERED(doc, items)</code>	A list
<code>DOCX.TABLE(doc, rows, header = TRUE)</code>	A table with grid lines
<code>DOCX.CODE(doc, text\$)</code>	A code listing
<code>DOCX.WRITE(doc, path\$)</code>	Writes the file
<code>DOCX.PLACEHOLDERS(path\$)</code>	The <code>{{name}}</code> holes of a template
<code>DOCX.FILL(src_path\$, dst_path\$, values)</code>	A copy with the holes filled from a map
<code>DOCX.READ(path\$)</code>	A map with the "paragraphs" and "tables" of a file
<code>DOCX.PARAGRAPHS(path\$), DOCX.TABLES(path\$)</code>	One of the two

PDFGEN: PDF files

Positions and sizes are millimetres from the top left corner of the page.

Function	What it does
<code>PDFGEN.DOC(opts = 0)</code>	A new document; <code>opts</code> may set "size", "margin" and more
<code>PDFGEN.ADDPAGE(doc)</code>	A new page
<code>PDFGEN.SETINFO(doc, title\$, author\$ = "", subject\$ = "")</code>	The document properties

Function	What it does
PDFGEN.ADDFONT(doc, family\$, path\$, style\$ = "")	Embeds a TrueType font
PDFGEN.USEFONT(doc, family\$, style\$ = "", size = 0)	Chooses a font, its style and size
PDFGEN.SETCOLOR(doc, r, g, b)	The colour of text
PDFGEN.SETFILL(doc, r, g, b), PDFGEN.SETDRAW(doc, r, g, b)	The colours of fills and lines
PDFGEN.SETLINEWIDTH(doc, mm)	The width of lines
PDFGEN.TEXTAT(doc, x, y, text\$)	Text with its baseline at x, y
PDFGEN.SETXY(doc, x, y), PDFGEN.POSY(doc)	Sets and reads the current position
PDFGEN.LN(doc, h)	To the left margin, h further down
PDFGEN.CELL(doc, w, h, text\$, align\$ = "L", border = 0, fill = 0)	One line of text in a box
PDFGEN.MULTICELL(doc, w, lineh, text\$, align\$ = "L")	A wrapped paragraph
PDFGEN.SPLITLINES(doc, text\$, w)	The lines a text breaks into
PDFGEN.FIT\$(doc, text\$, w)	A text cut to fit a width
PDFGEN.DRAWLINE(doc, x1, y1, x2, y2)	A line
PDFGEN.BOX(doc, x, y, w, h, style\$ = "D")	A rectangle, drawn, filled or both
PDFGEN.IMAGE(doc, path\$, x, y, w = 0, h = 0)	A JPEG picture
PDFGEN.TABLE(doc, rows, opts = 0)	A table across pages
PDFGEN.PAGECOUNT(doc), PDFGEN.PAGEWIDTH(doc)	The number of pages, the width
PDFGEN.WRITEFILE(doc, path\$)	Writes the file
PDFGEN.BUILD\$(doc)	The file as bytes, to attach to a mail

MAIL: messages

Function	What it does
<code>MAIL.MESSAGE(from\$, recipients, subject\$)</code>	A new message; recipients as text or array
<code>MAIL.PLAIN(msg, text\$)</code> , <code>MAIL.HTMLBODY(msg, html\$)</code>	The text, and an HTML version
<code>MAIL.CC(msg, recipients)</code>	Copies
<code>MAIL.HEADER(msg, name\$, value\$)</code>	Any other header
<code>MAIL.SETDATE(msg, date\$)</code> , <code>MAIL.SETID(msg, id\$)</code>	A fixed date and message id, for tests
<code>MAIL.ATTACH(msg, path\$, name\$ = "", mime\$ = "")</code>	Attaches a file
<code>MAIL.ATTACHDATA(msg, name\$, data\$, mime\$ = "")</code>	Attaches bytes held in a string
<code>MAIL.INLINE(msg, path\$, cid\$)</code>	A picture the HTML shows
<code>MAIL.BUILD\$(msg)</code>	The message as text
<code>MAIL.WRITEEML(msg, path\$)</code>	Writes an .eml file any mail program opens
<code>MAIL.SEND(msg, server)</code>	Sends through curl; a map with "ok"
<code>MAIL.COMMAND(msg, server, eml_path\$, config_path\$)</code>	The curl arguments that would send it
<code>MAIL.PARSE(eml\$)</code> , <code>MAIL.PARSEFILE(path\$)</code>	A message read back into a map
<code>MAIL.HEADERVALUE\$(parsed, name\$)</code>	One header of a read message
<code>MAIL.SAVEATTACHMENT(parsed, index, path\$)</code>	Saves an attachment
<code>MAIL.BAREADDRESS\$(address\$)</code> , <code>MAIL.ADDRESSNAME\$(address\$)</code>	The two parts of "Name <address>"

ICAL: calendars

Function	What it does
<code>ICAL.PARSEFILE(file_path\$)</code> , <code>ICAL.PARSE(src\$)</code>	Reads an .ics calendar
<code>ICAL.EVENTS(cal)</code>	Its events
<code>ICAL.PROP\$(cal, comp, prop_name\$, fallback\$ = "")</code>	A property such as "SUMMARY"
<code>ICAL.VALUES(cal, comp, prop_name\$)</code>	Every value of a property that repeats

Function	What it does
ICAL.ALLDAY(cal, comp)	Whether an event lasts whole days
ICAL.STARTAT(cal, comp), ICAL.ENDAT(cal, comp)	Start and end as DT instants
ICAL.OCCURRENCES(cal, comp, t_from, t_to)	The starts of a repeating event in a range
ICAL.EXPAND(cal, t_from, t_to)	Every event in a range as a DF table
ICAL.NEW(prodid\$)	A new calendar
ICAL.ADDEVENT(cal, summary\$, first_start, finish, zone\$ = "")	Adds an event
ICAL.ADDALLDAY(cal, summary\$, day_start, days = 1)	Adds an all-day event
ICAL.ADDALARM(cal, comp, minutes_before, text_in\$)	A reminder
ICAL.SETPROP(cal, comp, prop_name\$, value\$, params\$ = "")	Sets a property as written
ICAL.SETTEXT(cal, comp, prop_name\$, text_in\$)	Sets a text property
ICAL.TEXT\$(cal), ICAL.WRITEFILE(cal, file_path\$)	The calendar as text, or as a file

SCHED: when something runs

Function	What it does
SCHED.VALID(when\$)	Whether a schedule can be read
SCHED.PROBLEM\$(when\$)	What is wrong with it
SCHED.NEXTRUN(when\$, t, offset = 0)	The next run after t
SCHED.PREVRUN(when\$, t, offset = 0)	The last run before t
SCHED.UPCOMING(when\$, t, n, offset = 0)	The next n runs
SCHED.JOB(name\$, when\$, fn, opts = 0)	A job for the loop
SCHED.SERVE(end_at)	Runs the jobs until end_at, 0 for no end

A schedule is a cron line such as "30 7 * * mon-fri", a daily time such as "daily 06:00", or an interval such as "every 5 minutes".

TMPL: text with holes

Function	What it does
TMPL.RENDER\$(path\$, model)	A template file filled from a map
TMPL.RENDERSTR\$(src\$, model)	The same for a template in a text

`{{ name }}` is a hole, `{{{ html }}}` one whose text is not escaped, and `{% if %}` and `{% for x in items %}` repeat or leave out parts.

The further libraries

Each of these is used by a few recipes; the recipes named show how.

Library	Functions the recipes call	Recipes
CACHE	NEW, FETCH, PUT, HAS, SAVE, LOAD: values kept for a time, on disk	X05, X11
CLI	NEW, FLAG, OPT, ARG, MANY, CMD, PARSE, HELP\$, STATUS, DONE: command lines with options	M13, M14
DB	OPEN, CREATE, INSERT, INSERTMANY, FROM, WHERE, ORDERBY, ROWS, CLOSE: SQLite without SQL	X09
FUZZY	SCORE(a\$, b\$): how alike two names are, 0 to 100	M11
HTMLDOM	PARSE, FIND(doc, selector\$), ATTR\$, FLAT\$, ESCAPE\$: web pages read by CSS selector	X03, X11
JDWEB	GET, POST, PATCH, ASSETS, MOUNT, LISTEN, SERVE, REPLY, REPLY_JSON, PARAM\$, FETCH: small web servers on this computer	M08, X03, X15
JWT	SIGN\$, VERIFY, DECODE: signed tokens, behind AUDIT	X01, X14
LLMAPI	NEW(provider\$, model\$, api_key\$, base_url\$), SYSTEM, SET, JSON: language models, local or online	X02
METRICS	NEWGAUGE, SETVALUE, EXPOSE\$, RESET: numbers for monitoring	X13
OAUTH	NEW, SETOPT, DEVICESSTART, DEVICEPOLL, SESSION, GET, STORE, CACHEOF, EXPIRE: signing in to web services	X05
PKG	COMPARE(a\$, b\$): -1, 0 or 1 as version a\$ is older, the same or newer	X07

Library	Functions the recipes call	Recipes
POOL	CREATE, SPAWN, JOBS, RESULTS, TAKE_JOB, DONE, FAILED, GATHER, VALUE, SHUTDOWN: work spread over all cores	X08
PROPTTEST	FORALL(name\$, gen\$, prop): a property checked on many random cases	X12
QR	MATRIX, TEXT\$, TOPDF, INFO: QR codes	E15
REQ	NEW, HEADER, BEARER, TIMEOUT, RETRIES, GET, POST, PATCH, JSON, RAISE_FOR_STATUS, QUERY\$: HTTP with sessions	X04, X05, X11
RETRY	RUN(fn, arg, opts), ATTEMPT, TRANSIENT: trying again with growing pauses	X01, X02, X05, X07, X09, X11, X13
SEARCH	INDEX, ADD, REMOVE, QUERY, DOCCOUNT, WRITEFILE, READFILE: full-text search	X03
SECRET	TOKEN\$: random tokens and keys	X14
SVG	CHART(kind\$, w_px, h_px), SETOPT, LABELS, SERIES, RENDER, MARKUP\$, WRITEFILE: charts	M08, M09, X09, X15
TESTKIT	SUITE, EQ, NE, EQARRAY, ISTRUE, ISFALSE, NEAR, THROWS, SKIP, REPORT: the tests of every recipe	all
TEXTDIFF	TOKENS, OPCODES, RATIO, WORDDIFF\$: what changed between two texts	E14, M14
VALID	ISIBAN, IBAN\$: checks for bank account numbers	M11

The book's own modules

These live in `recipes/lib`, and the wizard copies them into your work folder.

Module	Functions	Used by
WORKCONF	See above	every recipe
OUTBOX	PUT\$(msg, cfg, file_name\$), PENDING(cfg), ASK\$(cfg), SEND_ALL(cfg, password\$), FOLDER\$, FROM\$, SERVER, SEND_ARGS: the mail rule of Chapter 4	M01, M03, M06, M08, M15, X02, X09, X10, X12, X13, X15

Module	Functions	Used by
VAULT	PUT(target\$, user\$, secret\$), GET\$(target\$), HAS, USER\$, DELETE, FIND\$(target\$, env\$, prompt\$), ASK\$(prompt\$): secrets in the Windows Credential Manager	OUTBOX, X01, X13, X14
JOBSTATE	JOBS(path\$), READ, WRITE, HEARTBEAT_AGE, and the paths FOLDER\$, JOBS_PATH\$, STATE_PATH\$, LOG_PATH\$, LOCK_PATH\$: the files of the job server	X01, X13, X14
AUDIT	APPEND(path\$, key\$, fields), LINES, ENTRIES, VERIFY(path\$, key\$): a signed record of every run	X01, X14
INPUTKIT	PARSE, PLAN_LINES, COUNTDOWN, RUN_STEPS, ACTIVATE, TYPETEXT, PRESS, CLICK: keyboard and mouse for programs without another way in, with an emergency stop	the bonus recipes

OUTBOX.ASK\$ takes the mail password from the Credential Manager when you stored it there as AutomateWork/mail, and otherwise asks for it with a prompt that prints stars.

Appendix C: Troubleshooting

These are the 25 messages readers of this book are most likely to meet, grouped by what you were doing when they appeared. Each one is shown as `jdBasic` or the recipe prints it, followed by what it means and what to do. A message longer than the page is broken over two lines here; on the screen it is one line. `jdBasic` prints the word *Error* and its number in color, and the rest as shown.

When a message is not in this list, read it from the end: the part after `at line` names the line where it happened. When `of` and a file name such as `CONF.jdb` follow, that line is in a shared library, and the words before it tell you more than the number. Those words usually name the file, the setting or the value that was wrong.

Installing and starting

1. Windows does not know `jdbasic`

```
'jdbasic' is not recognized as an internal or external command,  
operable program or batch file.
```

The command prompt cannot find `jdbasic.exe`. Either the folder of `jdBasic` is not on the `PATH`, or the prompt was opened before you added it. Close every command prompt and open a new one. If the message stays, repeat the four steps of *Installing jdBasic* in Chapter 2 and check that the folder you added is the one that holds `jdbasic.exe`. On a German Windows the same message starts with *Der Befehl "jdbasic" ist entweder falsch geschrieben.*

2. A library cannot be found

```
Error #99: Parse error at line 8: cannot load module 'WORKCONF'
```

The program imports a module that is in none of the places `jdBasic` looks: the program's own folder, a `modules` folder next to it, the folders in `JDBASIC_PATH`, `.jdbasic\lib` in your home folder, and the `lib` folder next to `jdbasic.exe`. The shared modules of the recipes, such as `WORKCONF` and `OUTBOX`, are copied into `.jdbasic\lib` by the setup wizard. Run the wizard once, or start the recipe from the copy in your work folder, not from the book's download. A module of the `jdBasic` libraries, such as `CONF` or `XLSX`, belongs in the `lib` folder of your `jdBasic` installation; unpack the release again if it is missing.

3. This release cannot compile

```
Native compilation not available (build with NATIVEC flag).
```

`jdbasic -c` needs a release of `jdBasic` with the native compiler. The forms release this book starts with has it, the smallest release does not. `jdbasic --version` lists *NativeC* among the features when it is there. Compiling also needs the Microsoft C++ build tools, as Chapter 2 explains under *From Script to Program*.

The settings file

4. No settings file

```
Error #99: No settings found at C:/Users/mia/Documents/AutomateWork/
config/work.conf. Run the setup wizard first, or name the file with
--config. at line 58 of WORKCONF.jdb
```

A recipe looks for `work.conf` after `--config` on the command line, then in the environment variable `AUTOMATEWORK_CONF`, then in the `config` folder of the work folder. Run the setup wizard, or give the file with `--config`. The line 58 of `WORKCONF.jdb` belongs to the shared module that reads the settings and says nothing about your program.

5. A folder with backslashes

```
Error #99: CONF.TOML C:/Users/mia/Documents/AutomateWork/config/work.conf line 5: "C:\Users\mia\Reports" is a Windows path in double quotes, where a backslash starts an escape; write the path with forward slashes, "C:/Users/mia", or in single quotes, 'C:\Users\mia' at line 48 of CONF.jdb
```

Inside double quotes a TOML file reads a backslash as the start of a special character, so "C:\Users\mia\Reports" cannot stay as it is written. The message names the line in `work.conf`, here line 5. Write the folder with forward slashes, "C:/Users/mia/Reports", which Windows understands as well, or in single quotes, 'C:\Users\mia\Reports', which keep every backslash.

6. A quote that is not closed

```
Error #99: CONF.TOML C:/Users/mia/Documents/AutomateWork/config/work.conf line 2: the text has no closing quote at line 48 of CONF.jdb
```

A value in `work.conf` starts with a double quote and has no second one on the same line. Open `work.conf` at the line the message names, here line 2, and add the quote at the end of the value; a folder that ends without its quote is the usual case.

7. A heading without its bracket

```
Error #99: CONF.TOML C:/Users/mia/Documents/AutomateWork/config/work.conf line 3: a section heading needs its closing bracket: [downloads_butler at line 48 of CONF.jdb
```

A heading in `work.conf` is missing its `]`: `[downloads_butler` instead of `[downloads_butler]`. Text without quotes gives a message of the same kind, *text needs quotes*: write `name = "Mia Example"`, not `name = Mia Example`. One mistake gives no message at all. A key with a different spelling, `min_age_hour` instead of `min_age_hours`, is read by nobody, and the recipe runs with its default instead of your value. Compare the part with Appendix D, which lists every key a recipe reads, and run the recipe with `--dry-run` to see the values it uses.

The wizard and the Task Scheduler

8. A schedule the wizard cannot read

```
WIZARD: cannot schedule "daily 8:30" for E01
```

The wizard understands daily HH:MM, weekdays HH:MM, weekly DAY HH:MM, every N minutes, every N hours, hourly, at logon and manual. The time needs two digits for the hour, so write daily 08:30. The day of weekly is a weekday name such as fri or friday.

9. The Task Scheduler refuses a task

```
could not schedule Downloads Butler: FEHLER: Ungültiger
Startzeitwert.
```

The wizard handed the schedule to the Task Scheduler, and Windows refused it. The text after the colon is Windows' own and comes in the language of your Windows; this one is from a German system and says that the start time is not valid. The usual cause is a time that does not exist, such as 25:00. Correct the schedule on the third page of the wizard and set up again. If the message speaks of missing rights, your company does not let you add tasks; ask IT, or start the recipes under the job server of X01, which needs only one task at logon.

10. The doctor finds a problem

```
no settings file at C:/Users/mia/Documents/AutomateWork/config\
work.conf
jdbasic not found; install it or name it in [paths] jdbasic
the shared module is missing: C:/Users/mia/.jdbasic/lib/workconf.jdb
E01: no scheduled task AutomateWork\E01 Downloads Butler
```

setup_console.jdb --doctor lists one line for each part of the setup that is not where the wizard put it. A missing settings file or shared module means the wizard has not run on this account, or the work folder moved. *jdbasic not found* means the path in [paths] no longer points to jdbasic.exe, usually after an update into a new folder. A missing task was removed in the Task Scheduler. Run the wizard again in each case; it keeps your answers.

Mail and the outbox

11. No mail server

```
Set [mail] server in work.conf first.
```

You ran a recipe with `--send`, and `work.conf` has no mail server. Add the `[mail]` part as Chapter 4 shows, or run the wizard again and fill in *Mail server* on its second page.

12. A message stays in the outbox

```
0 sent
  not sent: status.eml: curl: (7) Failed to connect to
mail.example.com:465 after 2022 ms: Could not connect to server
```

The recipe could not reach the mail server, so the message stays in the outbox for the next `--send`. The number after `curl:` tells the cause. (7) means no connection: a wrong server name or port, or a network that blocks it. (67) means the server refused your user name or password. (60) means the server's certificate could not be checked, which a company proxy often causes. Check the server and port with your IT department; many offices use port 465 with `smtps://` or 587 with `smtp://` and `starttls = true`.

13. No signing key for the audit trail

```
No signing key yet. Run with --init-audit first.
```

X14 checks the audit trail with a key kept in the Windows Credential Manager, and there is none for your account yet. Run `secrets_audit.jdb --init-audit` once. The key belongs to your Windows account; on another account or another computer the trail cannot be checked with it.

Pages in the browser

14. A port that is taken

```
Error #99: Port 8766 is taken. Set port in [find_anything]. at line 56
```

A recipe that shows a page on localhost needs a port of its own, and another program already listens on it. Often that program is the same recipe, started twice, for example once by its task at logon and once by hand. Close the other one, or set another number above 1024 for port in the recipe's part of `work.conf`.

Writing and changing code

15. A name `jdBasic` keeps for itself

```
Error #99: Parse error at line 1: expected variable name, got 'STEP'
```

`STEP` belongs to the `FOR` loop, and `jdBasic` does not accept it as the name of a variable. The same happens with other words of the language, such as `STOP`, `LINE`, `ON` or `CLS`, and with the names of builtin functions. Choose another name: `step_size`, `stop_time`.

16. A misspelt function

```
Error #22: Undefined function: PRNT at line 1
```

`jdBasic` knows no statement or function of that name. Check the spelling; `jdbasic --lint program.jdb` finds these before you run the program. A module function needs its module name in front, such as `WORKCONF.VALUE`, and its `IMPORT` line at the top.

17. A variable that was never declared

```
Error #20: Undeclared variable 'TOTAL' (OPTION "EXPLICIT") at line 2
```

With `OPTION "EXPLICIT"` at the top, every variable needs a `DIM` before its first use. Add `DIM total = 0` before the line, or check the name for a typing mistake: `totl` is a new variable, not a misspelt `total`.

18. An index past the end

```
Error #13: Array index out of bounds: 3 at line 2
```

The positions of a list start at 0, so a list of three items has the positions 0, 1 and 2. A loop over a list runs `FOR k = 0 TO LEN(items) - 1`. An empty list has no position at all; check `LEN(items) > 0` before reading `items[0]`.

19. A block without its end

```
Error #99: Parse error at line 4: expected 'END', got ''
```

An `IF` has no `ENDIF`, a `FUNC` no `ENDFUNC`, or a loop no `NEXT` or `LOOP`; for a `FOR` loop the message says expected `'NEXT'`. The line number is the end of the file, not the start of the block, which makes this the least helpful message of the list. Indent each block as the recipes do, and the block whose end is missing shows itself.

20. A program named like a library

```
Error #22: Undefined function: MAIL.MESSAGE at line 7
```

The function exists, but `jdBasic` loaded another file as the module. A program saved as `mail.jdb` next to your script is found before the library `MAIL`, because the program's own folder comes first. Give your own files names that no library uses, such as `my_mail.jdb`.

21. A wrong answer without a message

```
epo
0
```

These lines came from `PRINT MID$("Report", 1, 3)` and `PRINT INSTR("Report", "R")`. Positions in text start at 0 in `jdBasic`, as they do in lists, so `MID$(t$, 1, 3)` starts at the second letter, and `INSTR` answers 0 for a match at the very start. It answers -1 when there is no match, so test `INSTR(t$, x$) >= 0`, never `> 0`. Programs from other BASICs need this change more often than any other.

22. A setting that prints as NONE

```
value: NONE
```

A map was asked for a key it does not have. The answer is no value, and text made from it reads `NONE`. Check with `MAP.EXISTS(m, "key")` first, or read settings through `WORKCONF.VALUE` with a default, which is what the recipes do.

23. Arguments in the wrong order

```
Error #24: DATEADD: unknown unit ""; use Y, M, W, D, H, N or S at
line 1
```

`DATEADD` takes the unit first, then the number, then the date: `DATEADD("D", 1, day)`. With the date first, `jdBasic` reads the date as the unit. When a builtin answers with a message about a strange value, compare the order of its arguments with the reference.

24. Dates and numbers in other forms

```
Error #99: CDATE: "10/05/2026" is not a date; write YYYY-MM-DD or
DD.MM.YYYY, with HH:MM:SS after a space if needed at line 1
```

`CDATE` reads `2026-10-05` and the German form `05.10.2026`, each with a time if needed. A date with slashes can mean the 10th of May or the 5th of October, so `CDATE` stops with this message instead of guessing. Bring such a date into one

of the two forms first. Numbers need a decimal point: `VAL("12,5")` answers 12 without a message, so convert the form first with `REPLACE$(t$, ",", ".")`.

Compiling

25. Rules of the compiler

```
error at 2: STRICT: cannot assign DOUBLE to INTEGER 'MOOD'; wrap with  
CINT() to assign explicitly  
error at 1: undeclared variable 'TOTAL'
```

`jdbasic -c` is stricter than the interpreter. Every variable needs a `DIM`, and a variable keeps the kind of its first value: one that starts as 0 holds whole numbers, so a value with decimals needs `CINT(...)` or a start of 0.0. The number after `error at` is the line. A program that runs in the interpreter and fails here usually needs one of these two changes; the recipes of this book all pass both.

Appendix D: The Settings File

Every setting of the recipes lives in one file, `work.conf`, in the `config` folder of your work folder. This appendix lists every key a recipe reads. It is made from the recipes themselves each time the book is built, so it cannot fall behind them.

Each part of the file starts with its name in brackets. Text goes in double quotes, numbers and `true` or `false` do not, and a list is written in square brackets with commas between its items. Inside double quotes a backslash starts a special character, so write folders with forward slashes. A key you leave out takes the default shown here.

The shared parts

The setup wizard writes `[user]` and `[paths]`, and `[mail]` when you give a mail server. Several recipes read them. The last column names the recipes and shared modules that read a key.

`[user]`

Key	What it is	Read by
<code>name</code>	your name, for mail and documents	M15, X02, OUTBOX
<code>email</code>	your mail address	X11, X13, OUTBOX
<code>day_start</code>	when your working day starts, as HH:MM	X15
<code>day_end</code>	when it ends	X15

[paths]

Key	What it is	Read by
work	the work folder	X01, X02, X04, X06, X10, X14, X16, X17, X18
jdbasic	the jdbasic program the scheduled tasks start	X01, X06, X07, X10, X12

[mail]

Key	What it is	Read by
server	the mail server, such as <code>smtps://mail.example.com:465</code>	OUTBOX
user	your user name on the mail server	X14, OUTBOX
from	the sender of every message; without it your name and address from [user]	OUTBOX
outbox	the folder the messages wait in	OUTBOX
starttls	true when the server wants STARTTLS on port 587	OUTBOX
send_with	read by a program, not written by the wizard	OUTBOX

The mail password is never part of `work.conf`. The recipes ask for it when you send, or take it from the Windows Credential Manager under `AutomateWork/mail` (Chapter 4 and X14).

Chapter 3: the easy recipes

E01 Downloads Butler: [downloads_butler]

Key	What it is	Kind	Default
source	Folder to tidy	folder	"~/Downloads"
target	Folder to sort into	folder	"~/Downloads/Sorted"
min_age_hours	Leave files younger than (hours)	number	24

Key	What it is	Kind	Default
rules	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [downloads_butler] yourself to change it; the default is the program's.

E02 Morning Launcher: [morning_launcher]

Key	What it is	Kind	Default
every_day	Open every workday (programs, folders, web pages)	list	[]
mon	Open on Mondays as well	list	[]
fri	Open on Fridays as well	list	[]

E03 Batch Renamer: [batch_renamer]

Key	What it is	Kind	Default
folder	Folder whose files get dated names	folder	"~/Documents/Scans"
pattern	Pattern for the new name	text	"{date} {name}{ext}"
extensions	Only files with these extensions (empty: all)	list	["pdf", "jpg", "png"]

E04 One-Click Backup: [one_click_backup]

Key	What it is	Kind	Default
folders	Folders to back up	list	["~/Documents"]
target	Folder for the archives (best on another drive)	folder	"~/Documents/AutomateWork/backups"
keep	Archives to keep	number	10
max_file_mb	Leave out files larger than (MB)	number	200

E05 Break Reminder: [break_reminder]

Key	What it is	Kind	Default
start	Working hours start	time	"08:30"
end	Working hours end	time	"17:30"
every_ minutes	Remind me every (minutes)	num ber	50
workdays _only	Only Monday to Friday	bool	true
message	The reminder text	text	"Time for a break: stand up, stretch, drink some water."
state_ file	read by the program, not asked by the wizard (see below)		"~/Documents/AutomateWork/ logs/break_reminder.txt"

For a key marked *see below*, add it to [break_reminder] yourself to change it; the default is the program's.

E06 Shutdown Ritual: [shutdown_ritual]

Key	What it is	Kind	Default
journal	Journal file	file	"~/Documents/AutomateWork/ journal/journal.md"
summary_ weekday	Day of the weekly look back (1 Monday to 5 Friday)	numb er	5

E07 Time Tracker: [time_tracker]

Key	What it is	Kind	Default
folder	Folder for the time log and the weekly sheets	folder	"~/Documents/AutomateWork/time"

E08 Mail Templates: [mail_templates]

Key	What it is	Kind	Default
folder	Folder with your templates	folder	"~/Documents/AutomateWork/templates"
me	Your name under the mail	text	""
signature	The line before your name	text	"Kind regards"
values	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [mail_templates] yourself to change it; the default is the program's.

E09 Meeting Notes Starter: [meeting_notes]

Key	What it is	Kind	Default
calendar	Calendar file (.ics)	file	"~/Documents/AutomateWork/calendar.ics"
folder	Folder for the notes	folder	"~/Documents/Meeting notes"
lookahead_days	Look ahead (days)	number	7
utc_offset	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [meeting_notes] yourself to change it; the default is the program's.

E10 Duplicate Finder: [duplicate_finder]

Key	What it is	Kind	Default
folders	Folders to look through	list	["~/Documents", "~/Downloads"]
report	Excel report	file	"~/Documents/AutomateWork/duplicates.xlsx"
min_size_kb	Leave out files smaller than (KB)	number	1

E11 Disk Space Report: [disk_space]

Key	What it is	Kind	Default
root	Folder to measure	folder	"~"
warn_gb	Warn above (GB)	number	50
top	How many folders and files to list	number	10
report	File for the report	file	"~/Documents/AutomateWork/disk_space.txt"

E12 Birthday Reminder: [birthday_reminder]

Key	What it is	Kind	Default
contacts	Contact list (CSV)	file	"~/Documents/AutomateWork/contacts.csv"
calendar	Calendar file to write (.ics)	file	"~/Documents/AutomateWork/birthdays.ics"
days	Look ahead (days)	number	14

E13 Printable Week: [printable_week]

Key	What it is	Kind	Default
todo_file	Your to-do list (a text file)	file	"~/Documents/AutomateWork/todo.txt"

Key	What it is	Kind	Default
out_folder	Folder for the printable pages	folder	"~/Documents/AutomateWork/print"
days	Days on the page (5 or 7)	number	5
weeks_ahead	Which week (0 this week, 1 next week)	number	1

E14 Page Watcher: [page_watcher]

Key	What it is	Kind	Default
store	Folder for the copies of the watched pages	folder	"~/Documents/AutomateWork/watch"
page	read by the program, not asked by the wizard (see below)		[]

For a key marked *see below*, add it to [page_watcher] yourself to change it; the default is the program's.

E15 Guest Wi-Fi Card: [guest_wifi]

Key	What it is	Kind	Default
ssid	Name of the guest network	text	""
password	Its password	text	""
security	Security (WPA, WEP or nopass)	text	"WPA"
cards	Cards on the page (1 to 8)	number	8

Key	What it is	Kind	Default
hidden	The network is hidden	bool	false
out_file	File for the cards	file	"~/Documents/AutomateWork/print/guest_wifi.pdf"

E16 Form Filler: [form_filler]

Key	What it is	Kind	Default
values	The values, one per field, in the order of the form	list	[]
window	Title of the form's window (empty: the window in front)	text	""
countdown	Seconds to wait before typing	number	5
submit	Press Enter after the last field	bool	false

E17 Break Arcade: [break_arcade]

Key	What it is	Kind	Default
minutes	Length of a break in minutes	number	3
best_file	File that keeps your best score	file	"~/Documents/AutomateWork/arcade_best.txt"

E18 Office to PDF: [office_to_pdf]

Key	What it is	Kind	Default
source	Folder with the documents	folder	"~/Documents/AutomateWork/to_pdf"
target	Folder for the PDFs (empty: next to the documents)	folder	""
programs	Which documents: word, powerpoint, excel	list	["word", "powerpoint"]

Chapter 4: the medium recipes

M01 Invoice Generator: [invoice_generator]

Key	What it is	Kind	Default
sheet	The Excel file with the invoice rows	file	"~/Documents/AutomateWork/invoices.xlsx"
folder	Folder for the invoices	folder	"~/Documents/AutomateWork/invoices"
company	Your company's name	text	""
address	Your company's address (one line)	text	""
tab	Sheet tab with the orders (empty: the first)	text	""
prefix	Start of every invoice number	text	"RE-"
format	Invoice file: pdf or docx	text	"pdf"
vat_percent	VAT in percent	number	19

M02 Report Merger: [report_merger]

Key	What it is	Kind	Default
folder	Folder with one subfolder per month	folder	"~/Documents/AutomateWork/reports"
tab	Sheet to read in each report (empty: the first)	text	""
team	Names of the people who send a report	list	[]

M03 Mail Merge: [mail_merge]

Key	What it is	Kind	Default
templa te	Word template with {{placeholders}}	file	"~/Documents/AutomateWork/ templates/letter.docx"
contac ts	Excel file with the contacts	file	"~/Documents/AutomateWork/ contacts.xlsx"
folder	Folder for the letters	folde r	"~/Documents/AutomateWork/ letters"
subjec t	Subject of the mails	text	"Invitation to our customer day"
mail	Put a mail into the outbox for each letter	bool	true
tab	Sheet tab with the contacts (empty: the first)	text	""
file_ name	Name of each letter's file	text	"{{name}}"
body	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [mail_merge] yourself to change it; the default is the program's.

M04 Receipt Sorter: [receipt_sorter]

Key	What it is	Kind	Default
inbox	Folder where new receipts land	folder	"~/Documents/AutomateWork/receipts/inbox"
folder	Folder to file the receipts in	folder	"~/Documents/AutomateWork/receipts"
rules	Rules: word in the file name = category	list	["fuel = Travel", "rail = Travel", "hotel = Travel", "taxi = Travel", "office = Office", "lunch = Meals"]
kinds	read by the program, not asked by the wizard (see below)		["pdf", "jpg", "jpeg", "png"]

For a key marked *see below*, add it to [receipt_sorter] yourself to change it; the default is the program's.

M05 Shift and Vacation Planner: [shift_planner]

Key	What it is	Kind	Default
plan_file	The Excel file with the team plan	file	"~/Documents/Team/plan.xlsx"
calendar_file	Where to write the calendar (a shared folder)	file	"~/Documents/Team/team.ics"
max_away	Warn when more people than this are away	number	2
away_kinds	Kinds that mean away	list	["Vacation", "Sick", "Training"]
sheet	Name of the sheet in the workbook	text	"Plan"
title	Title of the plan	text	"Team plan"

M06 Friday Status Mail: [friday_status]

Key	What it is	Kind	Default
notes_folder	Folder with one notes file per person	folder	"~/Documents/Team/status"
to	Who gets the status mail	list	[]
subject	Subject ({week} becomes the week)	text	"Status {week}"
clear_done	Remove done notes after the mail (true or false)	bool	false
template	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [friday_status] yourself to change it; the default is the program's.

M07 Inbox Unpacker: [inbox_unpacker]

Key	What it is	Kind	Default
inbox_folder	Folder where you save mails (.eml)	folder	"~/Documents/AutomateWork/saved mail"
target_folder	Folder for the attachments	folder	"~/Documents/AutomateWork/attachments"
keep_inline	Also save pictures inside the text (true or false)	bool	false
done_folder	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [inbox_unpacker] yourself to change it; the default is the program's.

M08 Personal Dashboard: [dashboard]

Key	What it is	Kind	Default
todo_file	Your notes file with open items (- [] lines)	file	"~/Documents/AutomateWork/todo.txt"
downloads	Downloads folder	folder	"~/Downloads"
port	Port of the page on this computer	number	8765
folder	set in [time_tracker], read here too		"~/Documents/AutomateWork/time"
template	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [dashboard] yourself to change it; the default is the program's.

M09 Log Detective: [log_detective]

Key	What it is	Kind	Default
folder	Folder with the log files	folder	""
pattern	Which files (such as <i>.log</i>)	text	"*.log"
top	How many frequent errors to list	number	5
out_folder	Folder for the report and chart	folder	"~/Documents/AutomateWork/logs"

M10 Hot Folder: [hot_folder]

Key	What it is	Kind	Default
folder	Folder to watch	folder	"~/Documents/AutomateWork/hot"
done_folder	Folder for handled files	folder	"~/Documents/AutomateWork/hot/done"
rename	New names ({date}, {time}, {name})	text	"{date} {name}"
separator	Separator in CSV files	text	";"
settle_seconds	Seconds a file must rest before it is handled	number	60
rule	read by the program, not asked by the wizard (see below)		[]

For a key marked *see below*, add it to [hot_folder] yourself to change it; the default is the program's.

M11 Data Cleaner: [data_cleaner]

Key	What it is	Kind	Default
input	Export to clean (.csv or .xlsx)	file	"~/Documents/AutomateWork/export.csv"
separator	Separator in CSV files	text	";"
day_first	Dates like 05/10/2026 mean the 5th of October	bool	true
name_column	Column with customer names to compare	text	"Customer"
similar	How alike two names must be (0 to 100)	number	85
decimal	Decimal mark in the files (auto, or , or .)	text	"auto"
columns	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>
output	read by the program, not asked by the wizard (see below)		""

For a key marked *see below*, add it to [data_cleaner] yourself to change it; the default is the program's.

M12 Meeting Cost Meter: [meeting_cost]

Key	What it is	Kind	Default
calendar	Calendar file (.ics) to read	file	"~/Documents/AutomateWork/calendar.ics"

Key	What it is	Kind	Default
hourly_rate	Cost of one hour of work	number	60
currency	Currency	text	"EUR"
weeks	Weeks to look back	number	4
top	How many meetings the report lists	number	5
workbook	Workbook for the report	file	"~/Documents/AutomateWork/meeting_cost.xlsx"

M13 Snippet Tool: [snippet_tool]

Key	What it is	Kind	Default
folder	Folder for the snippets	folder	"~/Documents/AutomateWork/snippets"
values	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [snippet_tool] yourself to change it; the default is the program's.

M14 Contract Diff: [contract_diff]

Key	What it is	Kind	Default
reports	Folder for the change reports	folder	"~/Documents/AutomateWork/reports"
similar_percent	How alike (in percent) a changed paragraph must stay	number	60

M15 Timesheet for the Boss: [timesheet]

Key	What it is	Kind	Default
log	Another time log (empty: the one E07 writes)	file	""
folder	Folder for the timesheets	folder	"~/Documents/AutomateWork/timesheets"
boss	Who gets the timesheet (name and address)	text	""
folder	set in [time_tracker], read here too		"~/Documents/AutomateWork/time"

M16 Macro Player: [macro_player]

Key	What it is	Kind	Default
folder	Folder of your macros	folder	"~/Documents/AutomateWork/macros"
countdown	Seconds to wait before a macro starts	number	5

M17 Slide Builder: [slide_builder]

Key	What it is	Kind	Default
folder	Folder of your outlines and sheets	folder	"~/Documents/AutomateWork/slides"
out_folder	Folder for the presentations (empty: next to the outline)	folder	""
tab	Sheet of a workbook to read (empty: the first)	text	""

M18 Excel Refresh: [excel_refresh]

Key	What it is	Kind	Default
workbook	The report workbook	file	""
sheets	Sheets to make PDFs of (empty: all)	list	[]

Key	What it is	Kind	Default
pdf_folder	Folder for the PDFs	folder	"~/Documents/AutomateWork/reports"
mail_to	Mail the PDFs to (empty: no mail)	text	""

Chapter 5: the expert recipes

X01 Personal Job Server: [job_server]

Key	What it is	Kind	Default
jobs	The jobs file	file	"~/Documents/AutomateWork/config/jobs.toml"
folder	Folder for the state, the log and the lock	folder	"~/Documents/AutomateWork/logs"

X02 Inbox Assistant: [inbox_assistant]

Key	What it is	Kind	Default
inbox	Folder where you save mails (.eml)	folder	"~/Documents/AutomateWork/saved mail"
drafts	Folder for the drafts you approve	folder	"~/Documents/AutomateWork/drafts"
provider	Language model service (openai or anthropic)	text	"openai"
model	Model name	text	"gpt-4o-mini"
base_url	Own model server, such as http://localhost:11434/v1 (empty: the service)	text	""
max_mails	Most mails per run	number	20

X03 Find Anything: [find_anything]

Key	What it is	Kind	Default
folders	Folders to search in	list	["~/Documents"]
kinds	File kinds to read	list	["txt", "md", "csv", "docx", "pdf"]
index	Folder for the search index	folder	"~/Documents/AutomateWork/find"
language	Language of the documents (en, de or none)	text	"en"
port	Port of the search page on this computer	number	8766

X04 Ticket Sync: [ticket_sync]

Key	What it is	Kind	Default
tracker	Address of the ticket tracker, such as https://tickets.example.com	text	""
user	Your user name in the tracker	text	""
todo	Your to-do file (- [] lines)	file	"~/Documents/AutomateWork/todo.txt"
token_env	Environment variable that holds the token	text	"TICKETS_TOKEN"

X05 Calendar Bridge: [calendar_bridge]

Key	What it is	Kind	Default
client_id	Application id of the calendar app (from your IT)	text	""
tenant	Microsoft 365 tenant (organizations, or its id)	text	"organizations"
days	Days ahead to copy	number	14
label	What a busy block is called	text	"Busy (work)"

Key	What it is	Kind	Default
show_titles	Show meeting subjects (never for private ones)	bool	false
output	Calendar file your personal calendar subscribes to	file	"~/Documents/AutomateWork/calendar/work-busy.ics"
authority	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>
api	read by the program, not asked by the wizard (see below)		"https://graph.microsoft.com/v1.0"
token_file	read by the program, not asked by the wizard (see below)		"~/AutomateWork/calendar_token.json"

For a key marked *see below*, add it to [calendar_bridge] yourself to change it; the default is the program's.

X06 Work Cockpit: [work_cockpit]

Key	What it is	Kind	Default
tries	Starts of a failing automation before it counts as failed	number	1

X07 Tools for Colleagues: [tools_for_colleagues]

Key	What it is	Kind	Default
release	Release folder or web address colleagues update from	text	"~/Documents/AutomateWork/releases"
builds	Folder for the builds	folder	"~/Documents/AutomateWork/builds"

X08 Parallel Crunching: [parallel_crunching]

Key	What it is	Kind	Default
folder	Folder with the exports (CSV)	folder	"~/Documents/AutomateWork/exports"

Key	What it is	Kind	Default
reports	Folder for the reports	folder	"~/Documents/AutomateWork/reports"
separator	Separator in the CSV files	text	";"
workers	Workers (0: one per core)	number	0
timeout_seconds	Seconds one file may take	number	30

X09 Report Pipeline: [report_pipeline]

Key	What it is	Kind	Default
database	The bookings database (SQLite)	file	"~/Documents/AutomateWork/reports/bookings.db"
folder	Folder for the reports and the log	folder	"~/Documents/AutomateWork/reports"
to	Who gets the report (name and address)	text	""

X10 Let an AI Drive: [ai_drive]

Key	What it is	Kind	Default
allow	Recipes the agent may run for real (ids, such as E11)	list	[]
module	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [ai_drive] yourself to change it; the default is the program's.

X11 Web Harvester: [web_harvester]

Key	What it is	Kind	Default
folder	Folder for the targets, the CSV file and the cache	folder	"~/Documents/AutomateWork/harvest"
delay_seconds	Seconds between two requests to the same site	number	5
cache_hours	Hours a page read once is kept	number	12
targets	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [web_harvester] yourself to change it; the default is the program's.

X12 Tested Automations: [tested_automations]

Key	What it is	Kind	Default
recipes	Folder with the recipes to test	folder	"~/Documents/AutomateWork/recipes"
reports	Folder for the test reports and the log	folder	"~/Documents/AutomateWork/test reports"
to	Who gets the mail when a test fails (empty: you)	text	""

X13 Health Checks and Alerts: [health_checks]

Key	What it is	Kind	Default
alert_to	Who gets the alerts (empty: your own address)	text	""
late_minutes	Minutes a job may be late before it counts	number	30
server_minutes	Minutes the job server may stay silent	number	10
outbox_hours	Hours a mail may wait in the outbox	number	24

Key	What it is	Kind	Default
send_alerts	Send alerts at once (needs the mail password from X14)	bool	false
notify	Show new problems in a window (window or none)	text	"window"

X14 Secrets and Audit Trail: [secrets_audit]

Key	What it is	Kind	Default
summary_days	Days the audit summary looks back	number	7
names	read by the program, not asked by the wizard (see below)		[]

For a key marked *see below*, add it to [secrets_audit] yourself to change it; the default is the program's.

X15 Balance Score: [balance_score]

Key	What it is	Kind	Default
calendar	Your calendar exported as an .ics file	file	"~/Documents/ AutomateWork/calendar. ics"
target_hours	Hours of a normal week	number	40
meeting_share	Percent of the week in meetings that is still fine	number	40
weight_overtime	Points lost per hour of overtime	number	2
weight_meetings	Points lost per percent of meetings above that	number	1
weight_late	Points lost per hour worked outside your working hours	number	3
weight_mail	Points lost per mail written outside your working hours	number	1

Key	What it is	Kind	Default
port	Port of the page on this computer	number	8766
weeks	Weeks in the trend	number	8
folder	set in [time_tracker], read here too		"~/Documents/AutomateWork/time"
template	read by the program, not asked by the wizard (see below)		<i>worked out by the program</i>

For a key marked *see below*, add it to [balance_score] yourself to change it; the default is the program's.

X16 Desktop Robot: [desktop_robot]

Key	What it is	Kind	Default
job	The job file of the robot	file	"~/Documents/AutomateWork/robot/order_entry.toml"
countdown	Seconds to wait before the robot starts	number	5

X17 Meeting Briefing: [meeting_briefing]

Key	What it is	Kind	Default
calendar	Your calendar as an .ics file	file	"~/Documents/AutomateWork/calendar.ics"
mail_folders	Folders with saved mails (.eml)	list	["~/Documents/AutomateWork/saved mail"]
notes	Your notes file with open items (- [] lines)	file	"~/Documents/AutomateWork/todo.txt"
folder	Folder for the briefings	folder	"~/Documents/AutomateWork/briefings"
mails_per_meeting	Latest mails per meeting	number	3

Key	What it is	Kind	Default
use_model	Ask a language model for a summary	bool	true
provider	Language model service (openai or anthropic)	text	"openai"
model	Model name	text	"gpt-4o-mini"
base_url	Own model server (empty: the service)	text	""
mail_to_self	Put the briefing into the outbox as a mail to yourself	bool	false

X18 Spreadsheet to Web App: [sheet_app]

Key	What it is	Kind	Default
workbook	The Excel workbook	file	"~/Documents/AutomateWork/orders.xlsx"
sheet	The sheet (empty: the first)	text	""
port	Port of the app on this computer	number	8770
read_only	Columns the app shows but does not change	list	[]
backups	Folder for the copies before each save	folder	"~/Documents/AutomateWork/backups"

X19 Outlook Bridge: [outlook_bridge]

Key	What it is	Kind	Default
calendar_file	Calendar file to write (.ics)	file	"~/Documents/AutomateWork/calendar.ics"
mail_folder	Folder for the saved mails (.eml)	folder	"~/Documents/AutomateWork/saved mail"

Key	What it is	Kind	Default
days	Days of appointments, from today	number	1
mails	Newest mails to save	number	50

Appendix E: All 55 Recipes at a Glance

The minutes (*Min.*) are the time each recipe gives back in a normal week, as its page explains in the box at its end. The schedule is the one the wizard suggests; *by hand* means you start the recipe yourself or let the job server of X01 start it. The libraries are the ones the recipe imports besides its own modules and WORKCONF, which every recipe uses to read its settings.

Recipe	Min.	Schedule	Libraries	Page
E01 Downloads Butler	15	daily 18:30		99
E02 Morning Launcher	10	weekdays 08:00		106
E03 Batch Renamer	10	daily 18:00		111
E04 One-Click Backup	15	weekly fri 16:30		116
E05 Break Reminder	0	every 10 minutes		122
E06 Shutdown Ritual	20	weekdays end of day		126
E07 Time Tracker	30	weekly fri 16:00	XLSX	132
E08 Mail Templates	25	by hand	TMPL	140
E09 Meeting Notes Starter	20	weekdays 07:30	ICAL, DT, DF, DOCX	146

Recipe	Min.	Schedule	Libraries	Page
E10 Duplicate Finder	10	weekly fri 16:00	XLSX	152
E11 Disk Space Report	5	weekly mon 09:00		159
E12 Birthday Reminder	10	daily 08:00	DT, ICAL	165
E13 Printable Week	10	weekly fri 16:00	PDFGEN	171
E14 Page Watcher	20	daily 08:00	TEXTDIFF	179
E15 Guest Wi-Fi Card	5	by hand	QR, PDFGEN	186
E16 Form Filler	15	by hand	INPUTKIT	192
E17 Break Arcade	0	by hand	CONF	197
E18 Office to PDF	15	daily 12:00	OFFICEKIT	211
M01 Invoice Generator	10	by hand	XLSX, OUTBOX, MAIL, PDFGEN, DOCX	222
M02 Report Merger	15	by hand	XLSX	232
M03 Mail Merge	5	by hand	XLSX, MAIL, OUTBOX, DOCX	240
M04 Receipt Sorter	15	daily end of day	XLSX	249
M05 Shift and Vacation Planner	30	daily 07:00	XLSX, ICAL, DT	258
M06 Friday Status Mail	45	weekly fri 15:00	OUTBOX, MAIL, TMPL, DT	266
M07 Inbox Unpacker	30	daily 12:30	MAIL, DT	274
M08 Personal Dashboard	20	at logon	JDWEB, SVG, TMPL, OUTBOX	282
M09 Log Detective	30	weekdays 07:45	SVG	292
M10 Hot Folder	20	every 5 minutes	XLSX	301

Recipe	Min.	Schedule	Libraries	Page
M11 Data Cleaner	30	weekly mon 07:30	DT, FUZZY, VALID, XLSX	312
M12 Meeting Cost Meter	15	weekly fri 15:00	ICAL, DT, DF, XLSX	326
M13 Snippet Tool	20	by hand	CLI	335
M14 Contract Diff	30	by hand	CLI, DOCX, TEXTDIFF	344
M15 Timesheet for the Boss	15	monthly 1 08:00	OUTBOX, DOCX, MAIL	354
M16 Macro Player	30	by hand	INPUTKIT	366
M17 Slide Builder	30	by hand	XLSX, PPTX	375
M18 Excel Refresh	30	weekly mon 07:00	OUTBOX, MAIL	387
X01 Personal Job Server	30	at logon	JOBSTATE, LOGGER, VAULT, DT, SCHED, RETRY, AUDIT	402
X02 Inbox Assistant	60	by hand	MAIL, OUTBOX, LOGGER, LLMAPI, RETRY	416
X03 Find Anything	40	daily 12:30; at logon	JDWEB, LOGGER, SEARCH, DOCX, HTMLDOM	427
X04 Ticket Sync	30	by hand	LOGGER, REQ	440
X05 Calendar Bridge	25	every 2 hours	DT, LOGGER, OAUTH, CACHE, RETRY, REQ, ICAL	450
X06 Work Cockpit	20	by hand	CONF	463
X07 Tools for Colleagues	45	by hand	CONF, PKG, RETRY	476
X08 Parallel Crunching	10	by hand	POOL	488
X09 Report Pipeline	60	weekly mon 07:00	OUTBOX, LOGGER, SVG, DB, DF, DOCX, PDFGEN, MAIL, RETRY	499
X10 Let an AI Drive	20	by hand	CONF, OUTBOX, MAIL, LOGGER	513
X11 Web Harvester	45	weekdays 07:00	CONF, LOGGER, REQ, CACHE, RETRY, HTMLDOM	524

Recipe	Min.	Schedule	Libraries	Page
X12 Tested Automations	30	weekly sun 20:00	OUTBOX, MAIL, LOGGER	538
X13 Health Checks and Alerts	20	every 15 minutes	JOBSTATE, OUTBOX, VAULT, DT, SCHED, METRICS, MAIL, RETRY	552
X14 Secrets and Audit Trail	10	by hand	JOBSTATE, VAULT, AUDIT, SECRET	563
X15 Balance Score	15	at logon	OUTBOX, JDWEB, DT, ICAL, DF, SVG, TMPL	578
X16 Desktop Robot	60	by hand	LOGGER, INPUTKIT, CONF, IMG	591
X17 Meeting Briefing	45	weekdays 07:30	OUTBOX, MAIL, LOGGER, ICAL, DF, LLMAPI, RETRY, PDFGEN	604
X18 Spreadsheet to Web App	40	by hand	JDWEB, XLSX, LOGGER	619
X19 Outlook Bridge	20	weekdays 07:00	OUTBOX, ICAL, MAIL, OFFICEKIT	637

Level	Recipes	Minutes a week
Easy	18	235
Medium	18	420
Expert	19	625
All	55	1280

All recipes together come to 1280 minutes, about 21.3 hours a week. Nobody needs all of them; the ones you switch on are the number that counts, and Chapter 1 shows how to keep track of it.

Index

The page numbers refer to the printed book and its PDF. In the e-book, use the search of your reader instead.

A

- AI agent, 513
- ANDALSO, 49, 238, 342, 677
- API key, 416
- APPEND, 47, 680, 695
- ASYNC, 495, 675
- AUDIT, 694, 695, 731
- audit trail, 400, 402, 412, 521, 563, 576, 655, 701, 727, 731
- automation, 17, 670, and throughout

B

- backup, 116, 708, and throughout
- balance dividend, 12, and throughout
- BINREADER\$, 682
- byte order mark, 535, 536

C

- cache, 90, 91, 400, 534, 535, 694, 726, 731
- calendar, 450, 723, and throughout
- CATCH, 57, 64, 308, 685
- CDATE, 54, 681, 704
- checksum, 476, 484, 486, 497
- CHR\$, 66, 678
- CINT, 43, 678, 705
- CLI, 91, 694, 731
- cockpit, 463, 724, and throughout
- CODEC.SHA256\$, 156, 263, 684
- command prompt, 32, and throughout

- compiled program, 12, 22, 96, 398, 476, 496, 663
- CONF, 74, 675, 686, 698, 731
- CONF.GET, 75, 76, 675, 686
- CONF.TOML, 75, 76, 653, 686
- CONST, 675
- Credential Manager, 219, and throughout
- cron, 89, 412, 693
- CSV, 80, and throughout
- CSVREADER, 81, 82, 91, 138, 169, 289, 321, 361, 682
- curl, 562, 692, 701

D

- dashboard, 282, 717, and throughout
- data frame, 80, 81, 82, 509
- database, 79, 387, 400, 499, 500, 501, 509, 510, 511, 725
- DATEADD, 54, 80, 289, 509, 681, 704
- DATEDIFF, 54, 80, 138, 364, 681
- DB, 91, 694, 731
- DF, 80, 82, 88, 688, 692, 731
- DIM, 40, 42, 64, 238, 247, 280, 486, 676, 678, 685, 703, 705
- DIR\$, 67, 68, 70, 91, 103, 156, 271, 278, 682, 683
- DO WHILE, 51
- DOCX, 83, 84, 147, 227, 690, 731
- DOCX.FILL, 83, 84, 245, 690
- DOCX.PARAGRAPH, 83, 690
- DROP, 680

dry run, 3, and throughout
 DT, 79, 80, 91, 459, 587, 675, 681, 688, 692,
 731
 DT.FMT\$, 79, 80, 461, 688
 DT.PARSE, 79, 80, 278, 688

E

e-book, 2, 736
 .eml, 86, and throughout
 ENDSWITH, 45, 678
 environment variable, 73, and throughout
 ERRMSG\$, 57, 685
 Excel, 82, 387, 689, 721, and throughout
 EXITDO, 678
 EXPORT MODULE, 56, 675

F

FILE.COPY, 64, 68, 682, 683
 FILE.EXISTS, 67, 682
 FILE.ISDIR, 682
 FILE.MOVE, 64, 68, 69, 91, 103, 114, 278,
 682, 683
 FILE.STAT, 67, 156, 289, 308, 436, 682
 FILTER, 52, and throughout
 FOR, 3, 4, 50, 60, 64, 238, 678, 685, 702, 703
 FOR EACH, 50, 60, 64
 FORMAT\$, 44, 58, 59, 678, 679
 FORMAT_DATE, 54, 55, 80, 109, 124, 130,
 230, 247, 289, 300, 362, 681
 forms release, 38, 96, 464, 476, 514, 698
 FUNC, 55, 64, 91, 495, 675, 685, 703
 FUZZY, 91, 694, 731

G

GETENV\$, 683
 GRADE, 52, 156, 162, 169, 289, 298, 332, 350,
 680

H

health check, 399, 402, 412, 552, 563, 575,
 576, 649
 HTMLDOM, 91, 534, 694, 731
 HTTP.REQUEST, 684

I

ICAL, 82, 87, 88, 147, 166, 459, 616, 645, 692,
 731

ICAL.EXPAND, 88, 150, 331, 587, 692
 iCalendar, 146, 326
 .ics, 87, and throughout
 IF, 48, and throughout
 IMPORT, 56, 73, 74, 81, 91, 352, 438, 675,
 686, 702
 IN, 3, 4, 45, 47, 61, 64, 677
 INPUT, 59, 130, 131, 684, 685
 INPUTKIT, 695, 731
 INSTR, 45, 675, 678, 679, 704
 interpreter, 12, 22, 486, 488, 496, 705
 IOTA, 51, 228, 675, 680
 IT department, 192, 220, 366, 400, 450, 477,
 561, 564, 592, 701

J

JDBASIC_PATH, 73, 548, 551, 676, 698
 JDWEB, 91, 289, 684, 694, 731
 job server, 402, 722, and throughout
 JOBSTATE, 559, 695, 731
 JOIN, 46, 64, 678, 685
 Jonas, 26, and throughout
 JSON, 77, 289, 426, 436, 513, 521, 574, 587,
 684, 687, 694
 JSON.PARSE\$, 684
 JSON.STRINGIFY\$, 684
 JWT, 91, 694

K

KILL, 68, 69, 120, 682

L

LAMBDA, 52, 143, 156, 675
 language model, 91, 400, 416, 604, 605, 616,
 665, 666, 668, 722, 728
 LCASE\$, 45, 49, 238, 271, 678
 ledger, 29, 31, 672, and throughout
 LEFT\$, 44, 46, 67, 678
 LEN, 44, 46, 64, 130, 289, 497, 677, 678, 680,
 685, 703
 Lena, 399, and throughout
 LLMAPI, 91, 423, 694, 731
 localhost, 283, 290, 400, 425, 437, 618, 667,
 702, 722
 LOGGER, 77, 78, 521, 687, 731
 LOGGER.INFO, 77, 687

M

MAIL, 86, 352, 645, 691, 703, 731
MAIL.MESSAGE, 86, 238, 247, 256, 271, 692
MAIL.WRITEEML, 86, 692
MAP.DELETE, 681
MAP.EXISTS, 677, 681, 704
MAP.KEYS, 681
MAX, 298, 679, 680, 685
MCP, 513, 514, 522, 523
METRICS, 91, 559, 694, 731
Mia, 15, and throughout
MID\$, 44, 45, 46, 678, 704
MIN, 679, 680, 685
MKDIR, 68, 103, 682, 683, 687
MKTEMP\$, 65, 74, 397, 551, 662, 682, 683
MOD, 42, 59, 677

N

native compiler, 476, 663, 698
NONE, 41, 48, 676, 677, 681, 688, 704
NOW, 39, 54, 55, 80, 109, 230, 247, 681, 685

O

OAuth, 91, 450, 459, 461, 694, 731
OR ELSE, 49, 636, 677
OS.ARGS, 61, 683
OS.EXEC, 109, 412, 548, 683
outbox, 701, and throughout
OUTBOX.PUT\$, 228, 238, 271, 362, 392, 536

P

PATH, 32, 35, 341, 697
PATH.BASENAME\$, 682
PATH.DIRNAME\$, 247, 682
PATH.EXT\$, 66, 682
PATH.JOIN\$, 65, 67, 238, 247, 256, 682
PDF, 211, 690, 713, and throughout
PDFGEN, 85, 86, 173, 187, 228, 616, 690, 731
PDFGEN.TEXTAT, 85, 230, 690
personal data, 3, 23, 24, 417, 525, 605
PKG, 694, 731
POOL, 91, 694, 731
PRINT, 40, 352, 684, 685, 704
property test, 548
PROPTTEST, 91, 549, 694
PUSH, 47, 273, 680

Q

QR, 86, 91, 186, 187, 189, 191, 694, 731

R

recipe.toml, 89, 395, 397, 437, 464, 472, 521, 635, 660, 666
REGEX.FINDALL, 255, 263, 297, 340, 678
REGEX.MATCH, 678
REGEX.REPLACE, 678
REPLACE\$, 45, 278, 498, 678, 704
REQ, 91, 446, 684, 694, 731
retry, 25, 78, 91, 412, 459, 484, 559, 694, 731
RETURN, 55, 290, 324, 364, 561, 636, 675
RIGHT\$, 44, 59, 678
RMDIR, 68, 682
robots.txt, 524, 534, 536
ROUND, 58, 256, 679

S

SCHED, 89, 94, 412, 559, 693, 731
SCHED.NEXTRUN, 89, 693
schtasks, 657, 658, 659
SEARCH, 91, 694, 731
search index, 436, 694, 723
SECRET, 91, 694, 731
SELECT, 52, and throughout
SETENV, 683
setup wizard, 33, and throughout
SLEEP, 683
SMTP, 220, 701
SORT, 41, 47, 634, 680, 685
SPLIT, 46, 64, 678, 685
SQLite, 91, 500, 694, 725
STARTSWITH, 45, 618, 678
STARTTLS, 220, 701, 707
STR\$, 43, 678
SUB, 55, 675
SUM, 58, 83, 130, 156, 228, 271, 298, 362, 679, 685
SVG, 91, 289, 298, 499, 509, 694, 731

T

TAKE, 162, 298, 332, 680
Task Scheduler, 93, 700, and throughout
TESTKIT, 91, 92, 548, 694
TEXTDIFF, 91, 694, 731

THROW, 57, 238, 685
TICK, 208, 406, 534, 681, 685
time zone, 79, 278, 281, 334, 587, 589, 616
TMPL, 90, 91, 143, 270, 271, 289, 694, 731
TMPL.RENDER\$, 90, 271, 587, 694
TOML, 37, 62, 63, 74, 75, 76, 131, 593, 686,
699
TRIM\$, 45, 238, 678
TRY, 57, 63, 64, 308, 685
TXTREADER\$, 66, 271, 682
TXTWRITER, 66, 130, 682
TYPEOF, 676, 677

U

UCASE\$, 45, 52, 678
UNIQUE, 52, 322, 340, 680
UTF-8, 46, 70, 535, 536

V

VAL, 43, 322, 678, 685, 704
VALID, 91, 694, 731
VAULT, 576, 675, 684, 695, 731
VAULT.FIND\$, 576

W

WAITKEY\$, 684
web server, 283, 399, 484, 486
WEEKDAY, 681
Word, 83, 690, and throughout
work folder, 23, and throughout
work.conf, 62, and throughout
WORKCONF, 76, 687, and throughout
WORKCONF.FLAG, 62, 76, 687
WORKCONF.FOLDER\$, 76, 397, 687
WORKCONF.VALUE, 75, 76, 109, 194, 397,
687, 702, 704
worker pool, 675

X

XLSX, 73, 82, 83, 91, 138, 153, 438, 634, 689,
698, 731
XLSX.READ, 73, 82, 83, 228, 236, 263, 321,
384, 689
XLSX.SHEET, 82, 689

About the Authors

Achim Christ develops jdBasic, the language every program in this book is written in. jdBasic is a product of Computerwelt AI Solutions LLC.

His other books, in German, from the series *Die +n Bücher*:

- *Das +1 Buch: Der Notfall-Knopf für deine Jahres-Statistik*,
<<https://www.amazon.de/dp/B0GJG2782X>>
- *Das +2 Buch: Der Dopamin-Detox für deinen Ladebalken*,
<<https://www.amazon.de/dp/B0GHZTBRL1>>
- *Das +3 Buch: Die Jäger und Sammler des digitalen Nichts*,
<<https://www.amazon.de/dp/B0GJPG3YYJ>>
- *Das +4 Buch: Die Rückerstattung deiner Lebenszeit*,
<<https://www.amazon.de/dp/B0GL89D6BP>>

His

author

page: <<https://www.amazon.de/stores/Achim-Christ/author/B0GJS7VC2Q>>

jdBasic, its documentation and its releases:

<<https://github.com/AtomijD/jdBasic>>

Legal notice (Impressum): <<https://www.atomijd.onl/impressum.html>>

Claude is an AI model made by Anthropic. It wrote the chapters and the programs of this book with Achim Christ, and the build that turns them into this book, the e-book and the cover.