PACKAGE LAB



Package Lab
User Guide

From “what is a SCORM package?” to diagnosing a course nobody else can fix.

PACKAGE LAB	EDITION	PUBLISHED
0.3.0	1.2	22 November 2026

Contents

START HERE

About this guide

PART 1: FOUNDATIONS

1. Welcome
2. E-learning in ten minutes
3. Install, and build your practice kit
4. A tour of the window

PART 2: BEGINNER

5. Your first run
6. Reading the evidence
7. Your first diagnoses

PART 3: PRACTITIONER

8. Testing a theory
9. Macros: playing the learner
10. Saving and resuming
11. Browsers, devices and LMSs

PART 4: EXPERT

12. Going deeper
13. Delivering the answer
14. Working at scale
15. Thinking like an analyst

APPENDICES

A. Exercise answers
B. Quick reference
C. Troubleshooting Package Lab

Glossary

Change log

The Package Lab User Guide

From “what is a SCORM package?” to diagnosing a course nobody else can fix.

Edition 1.2, 22 November 2026, for Package Lab 0.3.0. See the change log for what changed.

This guide teaches Package Lab as a course, in order. Each chapter builds on the one before, and most have exercises on a set of practice packages that you build in chapter 3. Sixteen of the nineteen practice packages have one fault each; the other three are sound, and show you what “working” looks like. By the end you will have found every fault, and you will have the method to find the faults in real courses.

The guide and Help

Package Lab has two kinds of documentation, for two different moments:

- **This User Guide teaches you Package Lab.** Read it in order, do the exercises, and come out able to diagnose a course. It opens in a window of its own: **Help › User Guide** (⇧⌘G), or **User Guide** at the top of the Help panel. In the web interface, it opens in its own tab.
- **Help helps while you work.** It's the reference: every view, button and setting, one task to a page, in the inspector's Help panel beside your work. The ⓘ buttons and **What this means** on a finding open it.

The guide links to Help where you need the detail; those pages open in the guide's window, under a bar that leads back to your chapter.

Reading it in the app

The guide's window has the contents on the left, grouped in parts, and **On this page** on the right. Previous and next chapter are at the foot of each chapter. **Search the guide** searches every chapter.

- **Exercises.** Each *Try it* and *Exercise* heading has a **Done** box. The contents show how many of each chapter's exercises you've ticked, and the foot of the contents how many in all.
- **Your place.** The guide reopens at the chapter and the point you were reading.
- **The practice kit. Build practice kit**, under the contents, makes the practice packages in Package Lab's data folder. After that, the name of a practice package in the text (such as `01-works.zip`) is a button: choose it and the package opens in Package Lab. A macro file's name loads that macro into the **Macro** segment.

Reading it outside the app

- **One file:** every build writes `Package Lab User Guide.html` into `dist/` (and into the Windows zip), and `make guide` writes it on its own. It's the whole guide on one page, with the screenshots, for any browser.
- **A PDF:** every build also writes `Package Lab User Guide.pdf` beside it (and into the Windows zip), and `make guide-pdf` writes it on its own. It has a contents page, bookmarks for every chapter and heading, links between chapters that work, page numbers, a cover with the logo, version and edition, and each chapter on a new page.
- **On GitHub:** the chapters are plain Markdown, and read as they are.

These copies are for reading. To do the exercises, open the guide in the app (**Help › User Guide**): building the practice kit, and the package names that open each package, work only there.

Who it is for

Anyone who has to work out why an e-learning course doesn't work in an LMS: learning technologists, LMS administrators, course authors, support staff, developers. You don't need to know SCORM, JavaScript or how browsers work. Chapter 2 covers what you need, and later chapters add the rest as you go.

The chapters

Part 1: Foundations. What the tool is for, and what you need to know first.

1. Welcome: what Package Lab does, how it thinks, and how this guide works.
2. E-learning in ten minutes: packages, LMSs, SCORM, the API, the data model, frames and origins, cmi5 and xAPI.
3. Install, and build your practice kit: get Package Lab running and make the practice packages.
4. A tour of the window: the toolbar, sidebar, views, inspector and docs panel.

Part 2: Beginner. Run a course and read what happened.

1. Your first run: open a package, run it, read the verdict.
2. Reading the evidence: API calls, lookups, requests, errors, the boot path, the timeline and the code view.
3. Your first diagnoses: four broken packages, found from the findings.

Part 3: Practitioner. Test a theory without touching the course.

1. Testing a theory: one change at a time with settings, changed files, pins, answers and experiments.
2. Macros: playing the learner: recording, waits, clicks that follow their element, and macro files.
3. Saving and resuming: the save-and-resume test.
4. Browsers, devices and LMSs: Safari, iPads, blocked storage and LMS profiles.

Part 4: Expert. Find causes the findings can't name, and hand the answer on.

1. Going deeper: why it stalled, tracing, drill-down, server needs and a working copy.
2. Delivering the answer: corrected packages, reports without the course content, saved runs, comparing packages and checking a folder.
3. Working at scale: the command line, options files, and the web interface.
4. Thinking like an analyst: the method, the habits, the traps, cmi5 and xAPI, and a final challenge.

Back matter

- Appendix A: Exercise answers: what is wrong with each practice package, and the answers to the other *Try it* exercises. The *Check yourself* questions at the end of each chapter are answered in that chapter's text.
- Appendix B: Quick reference: the method, the milestones, symptoms, shortcuts and commands on one page.
- Appendix C: Troubleshooting Package Lab: when the tool, not the course, is the problem.
- Glossary: the terms this guide uses.
- Change log: what changed in each edition.

How long it takes

Part 1 takes about half an hour, most of it installing. Parts 2 and 3 take an afternoon each if you do the exercises. Part 4 is for dipping into as real courses bring you problems.

Making a website or a PDF

This section is for building a website of your own, or a PDF typeset with Pandoc and LaTeX instead of the one the build makes. The chapters are plain Markdown with relative links and no HTML, and combine into one document.

The order. This page, the chapters `01` to `15`, the appendices `A` to `C`, the glossary, then the change log. File names sort into that order, so a shell glob gives it.

One HTML file, with Pandoc 3 or later, from this folder:

```
pandoc --file-scope --lua-filter book.lua README.md [01]*.md appendix-*.md
glossary.md changelog.md --toc --embed-resources --standalone --metadata title="The
Package Lab User Guide" -o package-lab-guide.html
```

- `--file-scope` keeps each chapter's headings apart (every chapter has a *Try it* and a *Check yourself*), and turns links between chapters into links within the document. `book.lua` fixes the two kinds it misses: links to a whole chapter, and anchors that start with digits.
- `--embed-resources` puts the screenshots inside the file, so it can be moved.
- Links to the rest of Package Lab's docs (`../cli.md` and the like) stay as they are. In a standalone file they don't resolve; on a website that also publishes `docs/`, they do.
- Don't add `--number-sections`: the chapter titles carry their own numbers.

A PDF: the same command with `-o package-lab-guide.pdf --pdf-engine=xelatex -V mainfont="Helvetica Neue" -V monofont=Menlo`. XeLaTeX (from MacTeX or TeX Live) is needed for the ⌘, ⌥, ✓ and curly quotes the guide uses; the default `pdflatex` fails on them. Some wide tables will run close to the margin.

A website: MkDocs, Docusaurus and GitHub Pages take the folder as it is. List the pages in the order above, and publish the rest of `docs/` alongside it so the links to the reference work.

1. Welcome

The problem Package Lab solves

An e-learning course arrives as a ZIP file. Somebody uploads it to the LMS (the learning management system that hands courses to learners and records their progress). Then something goes wrong:

- the course doesn't open, or shows a blank page;
- it sits on a loading spinner for ever;
- it plays, but the LMS never records that anyone took it;
- learners come back and have to start again;
- it works in Chrome but not in Safari, or on a laptop but not on an iPad;
- it works in one LMS but not in the one your organisation uses.

The LMS usually says nothing useful. The authoring tool says the course is fine. The vendor asks for a screenshot. Everyone is guessing.

Package Lab stops the guessing. It plays the part of the LMS itself, runs the course in a browser of its own, watching it the way a programmer's debugger does, and records everything the course does: every file it loads, every message it sends to the LMS, every error, and what was on the screen. Then it turns that record into **findings**: what went wrong, the evidence, the likely cause and what to try.

Everything happens on your own computer. Package Lab itself sends nothing anywhere. The course runs in Package Lab's browser as it would in a learner's, so the course's own requests (fonts, analytics, its vendor's servers) still go out, and the **Network** view shows every one. Treat a course you don't trust as you would on any web page.

How Package Lab thinks

Four ideas run through the whole tool. Once you have them, its features make sense.

1. Read the files before running anything. A lot can be told from the ZIP alone: a manifest with the wrong name, a script that isn't in the package, a file name with the wrong capitals. Package Lab checks these the moment you open a package. These are *static* findings.

2. Then run it and record what really happens. Some problems only show when the course runs: a request to a server that isn't there, an error on line 4,000 of a script, a course that never looks for

the LMS. Package Lab launches the course in its own test LMS and records the run. These are *run* findings.

3. Measure how far it got. Every run is placed on a path of **milestones**: the page loaded, the scripts loaded, something showed, the course looked for the LMS, found it, started a session, read its saved data, wrote progress, saved it, and finished. The first milestone it missed is where to start looking. “How far did it get?” becomes a question with an answer, and a change that makes it get further is a change that helped.

4. Change one thing and run again. When you have a theory (“it can't find the LMS because the LMS is further away than the course looks”; chapter 2 explains how a course finds its LMS), you test it by changing one setting or one file in Package Lab, never in the real course, and running again. If it gets further, the theory was right.

That last idea is what separates an analyst from someone clicking buttons. Most of this guide is about it.

What Package Lab is not

- **It isn't your LMS.** Its test LMS follows the standards and can imitate the way several real LMSs host courses, but a run in Package Lab is a strong hint, not a verdict. Confirm every fix in the real LMS.
- **It isn't an authoring tool.** It can make small, mechanical corrections to a package and lets you try edits, but the lasting fix usually belongs in the authoring tool's export.
- **It doesn't judge the course's teaching.** It cares whether the course works and tracks, not whether it's good.

The ways to use it

Way	What it is	Where this guide covers it
Desktop app	A Mac (or Windows) window. Most of this guide uses it	Chapters 4 to 13
Command line	`package-lab` commands for quick checks, scripts and reports. Needs the Package Lab source	Chapter 14
Web interface	The desktop app in any browser, from another computer, iPad or phone. Needs the source too	Chapter 14

How this guide works

Each chapter has:

- **What you'll learn**, at the top;
- the explanation, with the buttons and views to use in **bold**;
- **Try it** exercises on the practice packages;
- **Check yourself**, a few questions at the end. If you can answer them, move on.

The practice packages are all made from one small sample course, *Fire Safety Basics*, with a single fault added to each. Their names describe the symptom a learner would report, never the cause. Finding the cause is the exercise. When you have a diagnosis, check it in the answers.

A note on words: this guide uses *course*, *package* and *content* almost interchangeably. Strictly, the **package** is the ZIP file, the **course** is what the learner sees, and the **content** is the course's own pages and scripts as they run. The glossary has every other term.

Check yourself

1. What is the difference between a static finding and a run finding?
2. Why does Package Lab care about the *first* milestone a run missed?
3. Where should you make a change to test a theory: in the course's files, or in Package Lab?

Next: E-learning in ten minutes.

2. E-learning in ten minutes

What you'll learn: what a package is, how a course and an LMS talk to each other, and the handful of ideas from browsers that explain most failures. If you already know SCORM well, skim the headings and move on.

The package

A course is a small website: HTML pages, JavaScript, stylesheets, images, video. A **package** is that website in a ZIP file, with a description of itself that an LMS can read.

For SCORM, the description is a file called `imsmanifest.xml`, at the top level of the ZIP. It says:

- which version of SCORM the course uses;
- the course's structure: its **items**, as a tree, each with a title;
- the **resources**: for each item, the page to launch (its `href`) and the files it needs.

An item that tracks the learner is a **SCO** (sharable content object). An item that is only a page to look at, such as a glossary, is an **asset**. A package often has one SCO; some have many.

Here is the core of the sample course you'll practise on, cut down (the namespace declarations and the second item are left out, so this extract isn't valid XML on its own):

```
<manifest identifier="com.thoughtasylum.toolbox.sample.scorm12" version="1">
  <metadata>
    <schema>ADL SCORM</schema>
    <schemaversion>1.2</schemaversion>
  </metadata>
  <organizations default="org">
    <organization identifier="org">
      <title>Fire Safety Basics (sample)</title>
      <item identifier="lesson" identifierref="lesson_res">
        <title>Lesson and quiz</title>
      </item>
    </organization>
  </organizations>
  <resources>
    <resource identifier="lesson_res" adlcp:scormtype="sco" href="lesson.html">
      <file href="lesson.html"/>
      <file href="scorm.js"/>
    </resource>
  </resources>
</manifest>
```

The LMS reads this, finds the item *Lesson and quiz*, and launches `lesson.html`.

Three things about the manifest break courses more than anything else, and Package Lab checks all of them as soon as you open a package:

- It must be called exactly `imsmanifest.xml`, in lower case, at the top of the ZIP (not inside a folder).
- Every file it names must be in the ZIP with exactly the same capitals.
- The SCORM version it declares must match the one the course's scripts actually use. Many LMSs choose which API to offer (`API` or `API_1484_11`) from the version the manifest declares, so a mismatch means the course looks for an API that isn't there.

The LMS and the API

When a learner launches a SCO, the LMS opens the launch page in the browser, usually inside a frame of its own page. The course and the LMS then talk through a JavaScript object called the **API**, which the LMS puts on one of its windows.

The course has to find it. It looks at its own window, then its parent frame, then the parent's parent, and so on up to the top, and sometimes at the window that opened it. The API has a different name in each version:

	SCORM 1.2	SCORM 2004
The object	`window.API`	`window.API_1484_11`
Start the session	`LMSInitialize("")`	`Initialize("")`
Read a value	`LMSGetValue(name)`	`GetValue(name)`
Write a value	`LMSSetValue(name, value)`	`SetValue(name, value)`
Save	`LMSCommit("")`	`Commit("")`
End the session	`LMSFinish("")`	`Terminate("")`
What went wrong	`LMSGetLastError()`	`GetLastError()`

Every call returns text: `"true"` or `"false"` for the actions, the value for a read. After a `"false"`, the error call returns a number that says why.

This explains a whole class of failures. A course that looks for `API` when the LMS only offers `API_1484_11` finds nothing. A course that searches three frames up, in an LMS that put the API four frames up, finds nothing. A course that never looks at all never talks to the LMS. Package Lab records every one of those lookups.

The data model

What the course reads and writes are named values, the **data model**. Their names start with `cmi.`. The ones you will see most:

Meaning	SCORM 1.2	SCORM 2004
Is this a fresh start or a return?	`cmi.core.entry` (`ab-initio` or `resume`)	`cmi.entry`
The bookmark: where the learner was	`cmi.core.lesson_location`	`cmi.location`
The course's own saved state (answers, progress)	`cmi.suspend_data`	`cmi.suspend_data`
Status	`cmi.core.lesson_status` (`passed`, `failed`, `completed`, `incomplete`…)	`cmi.completion_status` and `cmi.success_status`

Meaning	SCORM 1.2	SCORM 2004
Score	`cmi.core.score.raw`	`cmi.score.raw`, `cmi.score.scaled`
How the learner left	`cmi.core.exit` (`suspend`, `logout`, `time-out` or empty)	`cmi.exit` (also `normal`)
The learner's name	`cmi.core.student_name`	`cmi.learner_name`

Suspend data has a size limit: 4,096 characters in SCORM 1.2; in SCORM 2004 (3rd and 4th editions) an LMS must accept at least 64,000, and may accept more. Some LMSs set their own limit.

A healthy session

Put together, a course that works does this:

1. The LMS opens the launch page. The page loads its scripts and shows something.
2. The course looks for the API and finds it.
3. It calls `LMSInitialize` (or `Initialize`).
4. It reads `entry`, and if it is `resume`, reads the bookmark and suspend data and goes back to where the learner was.
5. As the learner moves, it writes the bookmark, suspend data, status and score, and commits now and then.
6. When the learner leaves, it sets `exit` to `suspend` if they haven't finished, commits, and calls `LMSFinish`.

Package Lab splits these six steps into ten **milestones**. Chapter 5 shows them on a real run.

Three ideas from browsers

Most courses that fail in an LMS but work on the author's computer fail for one of these reasons.

Paths and capitals. A file in the package is found by its path, relative to the page that asks for it. On a Mac or Windows, `js/App.js` and `js/app.js` are the same file. On the Linux servers most LMSs run on, they are different, and the wrong one is "not found". A path that starts with `/` (like `/api/config`) means the top of the *server*, not the package, so it only works on the server the course was built for.

Frames and origins. An LMS shows the course inside a **frame** (a page within a page), sometimes several deep, or in a new window. The course can reach the LMS's windows only if both come from

the same **origin**: the same scheme, host and port, such as `https://lms.example.com`. Many LMSs serve course files from a different host, a content server or CDN. Then the browser stops the course reaching the LMS's API at all, unless the LMS provides a bridge.

Storage and privacy. Browsers let a page keep data of its own: cookies, `localStorage`, `sessionStorage`, IndexedDB. Safari partitions or refuses that storage for a page shown inside another site's frame, and Chrome refuses it when third-party cookies are blocked; that frame is exactly where a course sits in an LMS. The details change between browser versions. A course that keeps its state there, rather than in `cmi.suspend_data`, can fail to start or forget everything in Safari or on an iPad.

cmi5 and xAPI, briefly

SCORM is old and widely used. Its successors are built on **xAPI** (also called Tin Can), where the course doesn't call an API object in the page. It sends **statements** ("Learner, Test *completed* Fire Safety") over the web to a **learning record store** (LRS).

- A **Tin Can package** has a `tincan.xml` instead of a manifest.
- A **cmi5 package** has a `cmi5.xml`. cmi5 adds rules on top of xAPI for how an LMS launches the course and what statements it must send, in what order.

Package Lab handles both, with its own LRS that checks every statement. Most of this guide uses SCORM, because that is what most courses are. Chapter 15 covers what changes.

Check yourself

1. A package's manifest is at `course/imsmanifest.xml` inside the ZIP. Will it work?
2. A SCORM 2004 course calls `LMSInitialize`. What will it find, and why?
3. Which data model values does a course need to read to resume where the learner left off?
4. The course loads `/scripts/player.js`. Why might that work on the authoring tool's website but not in an LMS?
5. Why might a course that keeps its progress in `localStorage` fail on an iPad?

Next: Install, and build your practice kit.

3. Install, and build your practice kit

What you'll learn: how to install the Package Lab app and make the practice packages the rest of this guide uses.

What you need

A Mac or a Windows PC, and the Package Lab app. That is all for every chapter except two optional extras: the WebKit (Safari engine) steps in chapter 11 and the command line in chapter 14 need Package Lab's source code, and chapter 14 explains them. Nothing else needs installing, and no Terminal.

This guide uses Mac keys: on Windows, Ctrl replaces ⌘, Alt replaces ⌥ and Shift replaces ⇧, so ⌥⌘I is Ctrl+Alt+I. Where something works only on a Mac, the guide says so.

Install the app

On a Mac:

1. Download `Package-Lab-mac-arm64.zip` from packagelab.thoughtasylum.com or the latest release. It is for Macs with Apple silicon (M1 and later). If your browser didn't unzip it, double-click the zip.
2. Drag **Package Lab** into your **Applications** folder.
3. Open it. The app is signed and notarised by Apple, so macOS opens it after asking once whether you want to open an app downloaded from the internet.

On Windows:

1. Download Package Lab for Windows from the same places: `Package-Lab-win-x64.zip` for most PCs, or `Package-Lab-win-arm64.zip` for Windows on Arm. It is a portable app: a zip with no installer.
2. Unzip it into any folder, a USB stick included.
3. Run `Package Lab.exe` in that folder. The Windows app isn't code-signed, so Windows may warn that it comes from an unknown publisher: choose **More info**, then **Run anyway**.

Nothing is written to the registry, and Package Lab doesn't make itself the app for `.zip` files. To uninstall it, delete the folder.

To see whether there is a newer version, choose **Help › Check for Updates...**. Package Lab asks GitHub only when you choose it, and offers to open the download page.

Open Package Lab

Open **Package Lab** from Applications or Launchpad on a Mac, or run `Package Lab.exe` on Windows. On a Mac, you can also right-click a course's `.zip` in Finder and choose **Open With › Package Lab**.

You should see a window that says what Package Lab does and lists common symptoms. That is the **start screen**, and chapter 4 explains it. Leave the window open.

Build the practice kit

The practice packages are all made from one small SCORM 1.2 sample course, *Fire Safety Basics*: four pages and a three-question quiz, which bookmarks the learner's page, saves their answers, and records a score.

Build it in the app, on a Mac or Windows: open this guide in its own window (**Help › User Guide**, ⇧⌘G) and choose **Build practice kit**, under the contents. The packages go in the `practice` folder of Package Lab's data folder, and **Show folder** opens it. From then on, every practice package name in the guide's text is a button that opens that package in Package Lab. Build the kit again at any time to get fresh copies.

These are the packages:

Package	What a learner would report	First used in
`01-works.zip`	Nothing: it works. Use it to learn what "good" looks like	Chapter 5
`02-blank-page.zip`	"The course is just a white page"	Chapter 7
`03-not-tracked.zip`	"I finished it, but the LMS says I never started"	Chapter 7
`04-loading-forever.zip`	"It says Loading and never gets past it"	Chapter 7
`05-error-in-the-console.zip`	"The browser console shows an error. Does it matter?"	Chapter 7
`06-stuck-connecting.zip`	"It says *Connecting to the LMS*… for ever, and nothing is recorded"	Chapter 8
`07-works-on-the-platform.zip`	"It worked in the authoring tool's preview, but not in our LMS"	Chapter 8
`08-forgets-its-place.zip`	"When I go back, I'm at the start again"	Chapter 10

Package	What a learner would report	First used in
`09-starts-over.zip`	"It asks me where I was, then starts from page 1 anyway"	Chapter 10
`10-fails-in-safari.zip`	"Fine on my PC, broken on my iPad"	Chapter 11
`11-fails-in-some-lmss.zip`	"Works in one LMS, stuck on *Connecting…* in another"	Chapter 11
`12-waits-forever.zip`	"No errors anywhere, but nothing is recorded"	Chapter 12
`13-keeps-looking.zip`	"It never connects, but the vendor says it's fine"	Chapter 12
`14-next-button-late.zip`	"Our automated test clicks Next and nothing happens"	Chapter 9
`15-button-moves.zip`	"The macro works on my screen but not on the test machine"	Chapter 9
`16-cmi5-never-passes.zip`	"The LMS shows the lesson completed, but never passed"	Chapter 15
`17-tincan-answer-missing.zip`	"The completion arrives, but the quiz answer never shows in our reports"	Chapter 15
`18-lost-in-some-lmss.zip`	"Fine in our test LMS, never connects in Moodle or Canvas"	Chapter 11
`19-forgets-its-answers.zip`	"It remembers my page, but not my answers"	Chapter 11

Packages 14 and 15 come with a macro file each (`14-next-button-late.macro.json`, `15-button-moves.macro.json`), and their course has nothing wrong with it: the exercise is the macro. Packages 16 and 17 are cmi5 and Tin Can (xAPI) courses, for the last chapter.

Each package has exactly one fault, except `01-works.zip`, `14-next-button-late.zip` and `15-button-moves.zip`, whose courses are sound. The names tell you the symptom, never the cause. Diagnose with Package Lab's findings and views first, as you would on a real course where the code is 40,000 lines of minified JavaScript; read the code when an exercise sends you there, and not before. When you have a diagnosis, check it in the answers.

Where Package Lab keeps things

On a Mac, Package Lab keeps its data in `~/Library/Application Support/Package Lab`; on Windows, in `%APPDATA%\Package Lab`, unless there is a folder named `Package Lab Data` beside

`Package Lab.exe`, which it then uses instead, so that nothing is left on the PC. **Help › Show Data Folder** opens whichever it is:

- `runs/`: every run, saved as JSON;
- `macros/`: macros you save by name;
- `discoveries/`: the results of **Server needs**;
- `practice/`: the practice kit, when you build it from the guide;
- `settings.json`: the app's own settings.

Saved runs hold the course's scripts and the values it stored. Treat them as you would the course itself.

Check yourself

1. Where do you build the practice kit, and where does it go?
2. Which practice package should you open first, and why?
3. Where would you find a run you did yesterday?

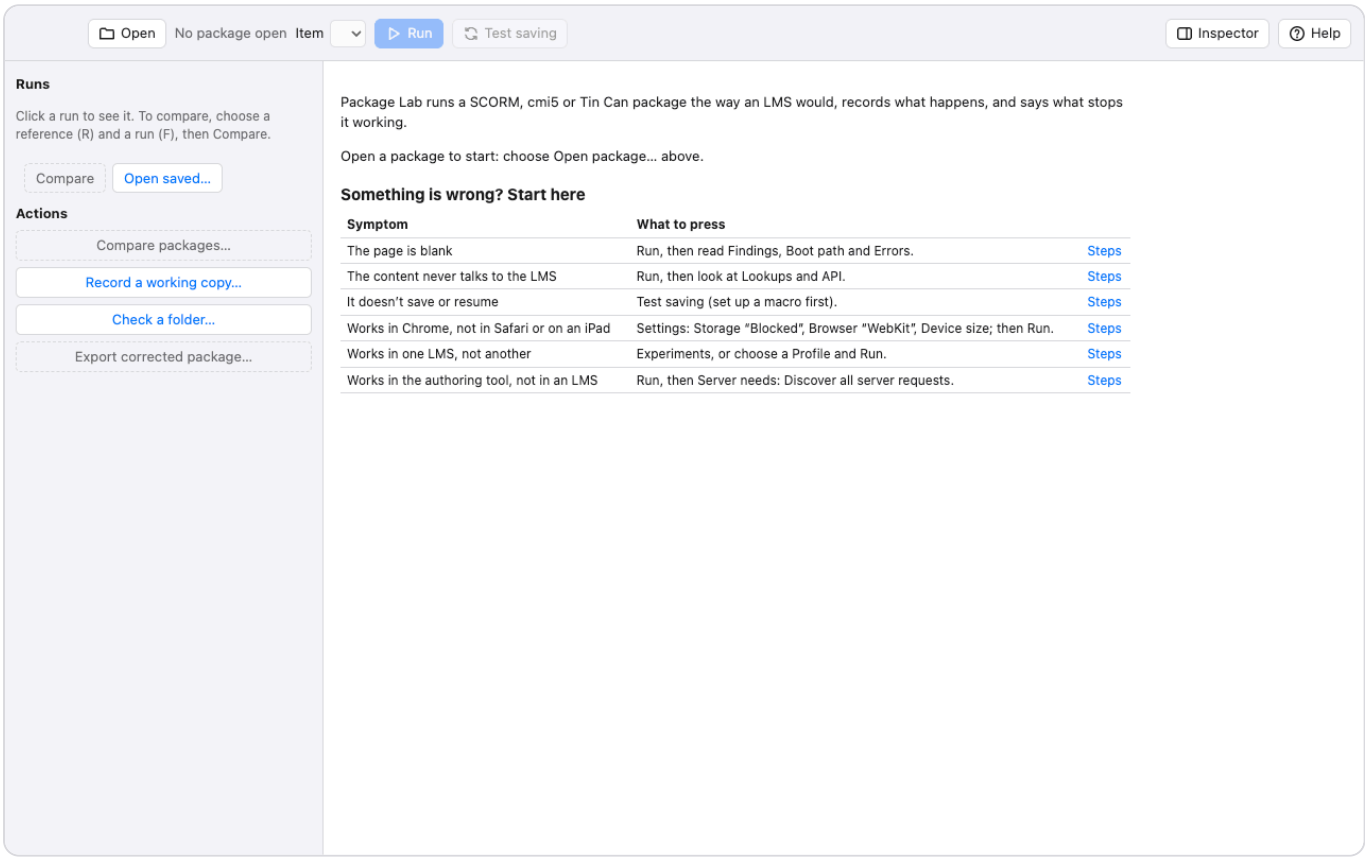
Next: A tour of the window.

4. A tour of the window

What you'll learn: where everything is, what each part of the window is for, and how to get help from inside the app. Don't try to remember it all: come back here when a later chapter mentions something you can't find.

The start screen

With no package open, the window explains what Package Lab does and lists the usual symptoms, each with what to press and a **Steps** link to the recipe for it.



When you have a real course with a real problem and no idea where to start, start here. Find the symptom, do what it says, and read the steps.

Opening a package

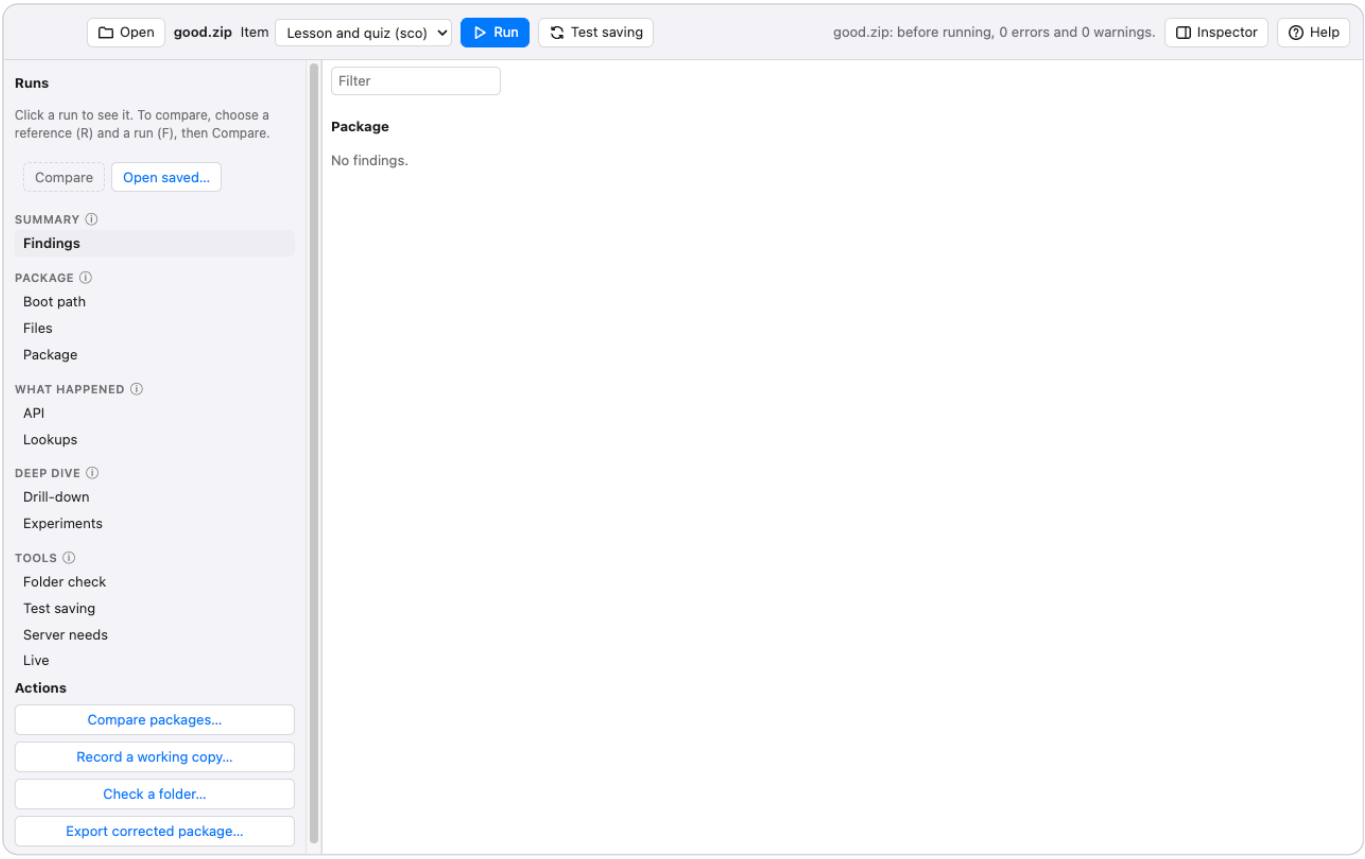
Any of these opens a package:

- **Open package...** in the toolbar, or **File › Open Package...** (⌘O);
- drag the `.zip` onto the window (it says **Drop to open**), or, on a Mac, onto the Dock icon;
- on a Mac, right-click the `.zip` in Finder and choose **Open With › Package Lab**;

- **File › Open Recent**.

Package Lab reads the ZIP in memory and never unpacks it to disk. The static findings appear at once.

Try it. Open `01–works.zip` from your practice folder.



The three parts of the window

The toolbar (along the top)

From left to right:

- **Open** and the package's name. (The **File** menu calls it **Open Package...**.)
- **Item**: which item of the package to launch. The first SCO is chosen for you.
- **Run** (⌘R), which becomes **Stop** (⌘.) while a run goes.
- **Record** (only while a run goes): records your clicks as a macro (chapter 9).
- **Show in Package Lab** (only while a run goes): shows the course window inside Package Lab, in the **Live** view.
- **Test saving** (⇧⌘R): the save-and-resume test (chapter 10).
- The **summary chip**, beside Run: a short list of the settings you've changed from their defaults, such as “SCORM 2004, Moodle, iPhone”. It's hidden when everything is at its

default. Click it to see those settings. **Glance at it before every run**: a setting left over from an earlier test is the most common reason a run surprises you. Most settings are remembered even after you quit: **API**, **Profile**, **Frame depth**, **In a new window**, **Stop after**, **Record caught errors**, **Storage**, **Browser**, **Device size** and **Trace the code**. The **Item**, **Resume last run's data**, pins, answers and the macro are not.

- The **status line**: what Package Lab is doing, and the count of errors and warnings.
- **Inspector** (⌥⌘I) and **Help**, at the right.

The sidebar (left)

At the top, **Runs**: every run of this session in the order you made them, newest at the bottom (and selected), and any saved runs you open. Click one to show it. With no run selected, the views show the package as read from its files; with a run selected, they show that run. This matters: the same view, **Boot path** say, shows different things before and after a run. Once you've run a package, a run stays selected; to see the files-only views again, open the package again (**File › Open Recent**).

Under **Runs** are **Open saved...** (runs from earlier sessions, or from the command line) and the **R** and **F** marks, which set a reference run and a failing run to compare (chapter 12).

Below that, the **views**, in five groups. You'll meet each one in the chapter shown.

Group	Views	What they answer	Chapter
Summary	Findings, Timeline	What went wrong? What happened when?	5, 6
Package	Boot path, Files, Package	What's in the package, and how should it load?	6, 8
What happened	API, xAPI, Network, Errors, Console, Frames, Lookups, Scripts	What did the course actually do?	6
Deep dive	Execution, Drill-down, Experiments	Why did it do that?	8, 12
Tools	Compare, Package diff, Folder check, Test saving, Server needs, Live	Tests and comparisons you start	9, 10, 12, 13

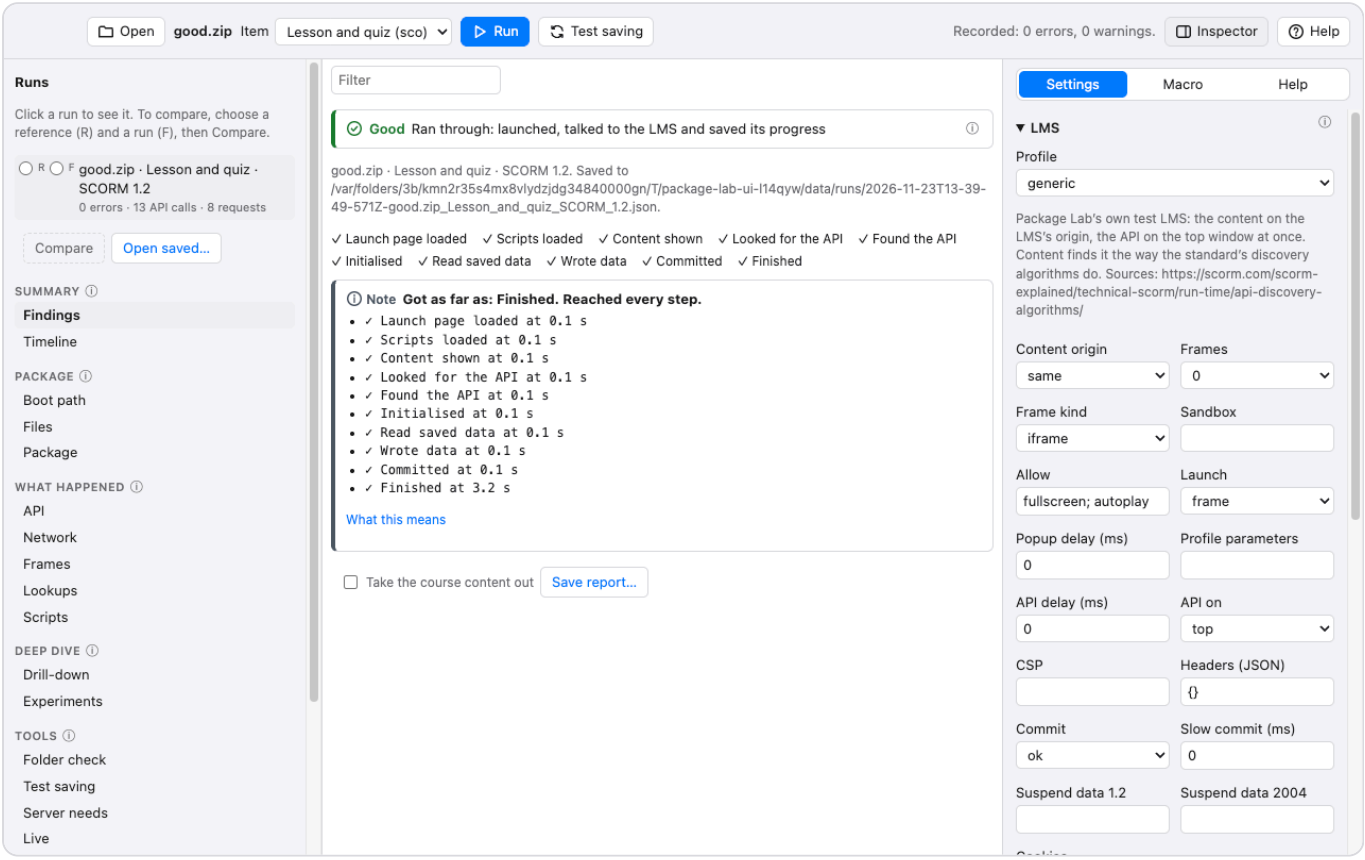
A view with nothing to show is left out of the list (the xAPI view appears only for cmi5 and Tin Can runs, for example). ⌘1 to ⌘5 jump to the first view of each group, and ↑ and ↓ move through them. The **Filter** box above a view narrows whatever table it shows.

At the bottom, **Actions**: **Compare packages...**, **Record a working copy...**, **Check a folder...** and **Export corrected package...** (chapter 13).

The inspector (right)

Inspector in the toolbar (⌥⌘I) opens a panel with three segments:

- **Settings**: everything the next run will use, in four groups: **LMS**, **Browser and device**, **Recording** and **Advanced**. Chapters 8 to 11 cover them.
- **Macro**: the clicks and keys Package Lab plays into the course (chapter 9).
- **Help**: the docs, including this guide.

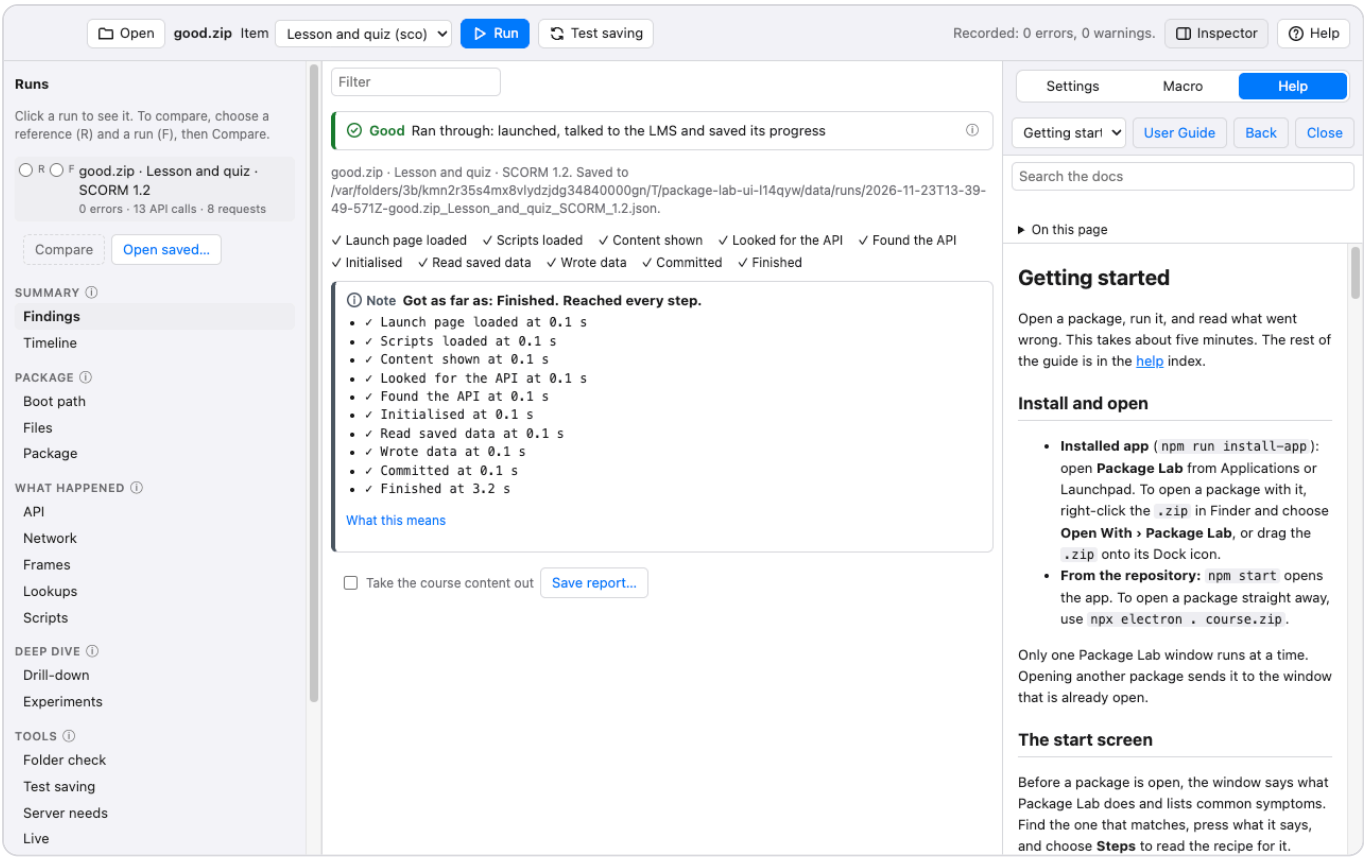


Drag the inspector's left edge to make it wider. It remembers its width and which segment was open.

Getting help inside the app

There are two kinds of help, for two different moments.

Help, in the toolbar (or ⇧⌘H), is for while you work. It opens the reference in the inspector, beside the window: what each view, button and setting does. You can search it as you type, and **On this page** lists the headings of the page you're reading. The small ⓘ beside a settings group, a view group and the verdict opens the part of Help that explains it, and **What this means** on a finding opens its explanation.



The User Guide, which you're reading, is for learning. It has a window of its own (**Help › User Guide**, ⇧⌘G, or **User Guide** at the top of the Help panel), so you can keep a chapter open beside Package Lab while you do its exercises. See About this guide for what its window does.

Both are inside the app, so they work offline.

The menus

On Windows the same menus are in the window's own menu bar, with **Package Lab** first.

Menu	What's there
Package Lab	**Settings...** (⌘,): Appearance (System, Light or Dark) and the web interface's port
File	Open, Open Recent, Compare Packages, Check a Folder, Open Saved Run, Record a Working Copy, Export Corrected Package (⌘E)
Run	Run (⌘R), Stop (⌘.), Record Macro (⌥⌘R), Test Saving (⇧⌘R)
View	Show Inspector (⌥⌘I), Reload (⇧⌥⌘R)
Help	User Guide (⇧⌘G), Show Help (⇧⌘H), Help Topics, Recipes, Findings, Show Data Folder

The shortcuts worth learning now

Keys	Does
⌘O	Open a package
⌘R	Run
⌘.	Stop
⌥⌘I	Show or hide the inspector
⌘1 to ⌘5	Jump to a view group
⇧⌘H	Help
⇧⌘G	User Guide

Check yourself

1. With no run selected, what do the views show?
2. Where do you look to see which settings the next run will use, without opening the inspector?
3. You can't see the **xAPI** view in the sidebar. Is something wrong?
4. How do you find the explanation of a finding you don't understand?

Next: Your first run.

5. Your first run

What you'll learn: how to open a package, run it, use it as a learner would, and read the verdict. You'll use `01-works.zip`, which has nothing wrong with it, so you learn what a good run looks like before you meet a bad one.

Step 1: open the package and read the static findings

Open `01-works.zip` (⌘O, or drag it onto the window). Package Lab reads the files and fills **Findings** at once.

For this package the list is empty, and the status line says there are no errors or warnings. That means the manifest is where it should be, every file it names is in the ZIP with the right capitals, the scripts use the same SCORM version the manifest declares, and every file the launch page loads is there.

Look at the **Item** menu in the toolbar. It shows *Lesson and quiz*, the first SCO. The package has a second item, *Glossary*, which is an asset: it doesn't track, so it isn't chosen.

Step 2: choose how the run ends

Open the inspector (⌥⌘I) on **Settings**, and find **Stop after** in the **Recording** group. It has two kinds of choice:

- **I stop it**: the run lasts until you press **Stop** or close the course window. Use this when you want to use the course yourself.
- **A time** (10 s to 5 min): Package Lab stops by itself and analyses the run. Use this for a quick look, or when the course is broken and won't get anywhere anyway.

Choose **I stop it** for now. Leave every other setting as it is.

Step 3: run it and use it

Choose **Run** (⌘R). The course opens in a window of its own, with Package Lab playing the LMS. Back in the main window, the **API** view fills in as the course talks to the LMS.

The course's first line greets you by name (the test LMS's learner is *Learner, Test*) and shows what it read from the LMS: entry `ab-initio`, status `not attempted`. Then, as a learner would:

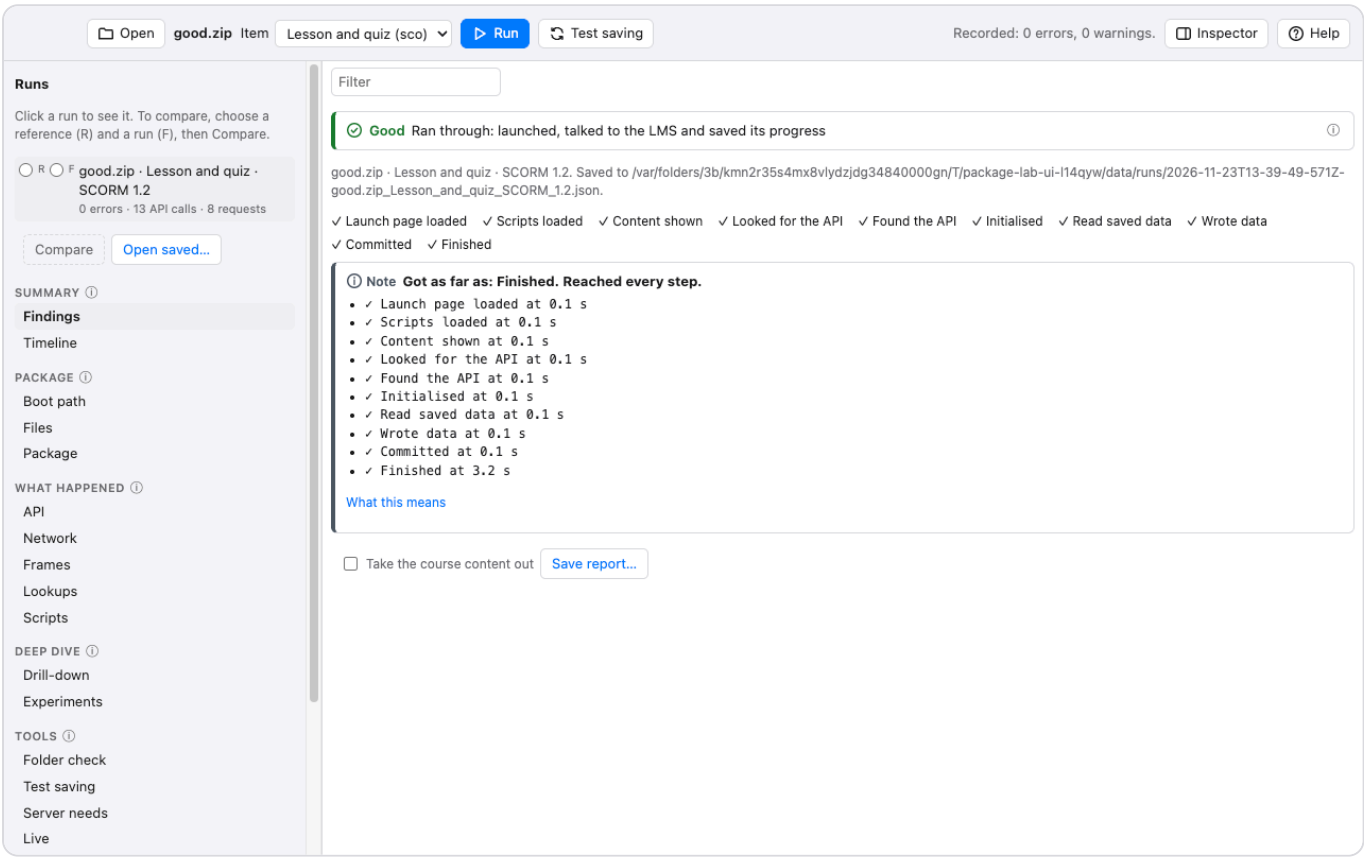
1. Choose **Next** three times, to reach the quiz on page 4.

2. Answer the three questions on the quiz page, and choose **Submit answers**.
3. Choose **Save and exit**.
4. Close the course window, or choose **Stop** in Package Lab.

Package Lab now collects what it recorded, works out the findings, saves the run, and selects it in **Runs**.

Step 4: read the verdict

Findings now starts with a banner. It has one word and one sentence:



- The word is **Problem**, **Warning** or **Good**: the worst finding of the run. **Good** means no problem and no warning.
- The sentence says how far the run got. “Ran through” means it reached every milestone. Otherwise it says “Got as far as “…”, but didn't reach “…”.”

Under the banner is the **Milestones** strip, then the findings, worst first.

The milestones

Each of the ten steps has a tick or a cross. Hover over one to see what showed it.

#	Milestone	Reached when
1	Launch page loaded	The launch page came back successfully
2	Scripts loaded	Every script the page names was requested, and none failed
3	Content shown	Some frame showed at least 20 characters of text, or an image or other media
4	Looked for the API	The content searched for the SCORM API
5	Found the API	It found one
6	Initialised	`LMSInitialize` or `Initialize` returned true
7	Read saved data	It read the entry, the location or the suspend data
8	Wrote data	It set the location, suspend data, status, score or completion
9	Committed	A commit returned true
10	Finished	`LMSFinish` or `Terminate` returned true

These are the healthy session from chapter 2, step by step. Learn them: from now on, “where did it stop?” is the first question you ask of every run.

The findings

Each finding is a card with a severity (**Problem**, **Warning** or **Note**), a title, an explanation, the **evidence** (a file and line, or what happened) and **Try:**, what to do. **What this means** opens the full explanation in the docs panel.

The three severities have other names outside the window, and you'll meet both:

In the window	In the command line, reports, the folder check and the findings reference
Problem: this will stop it working	**error** (`ERROR`)
Warning: it may stop it working, or has a side effect worth knowing	**warning** (`WARNING`)
Note: context	**info** (`INFO`)

This guide says *Problem* when you're in the window, and *error* when it quotes the command line.

Step 5: see what a timed run says

Now do it the quick way. Set **Stop after** to **10 s**, choose **Run**, and don't touch the course window. After ten seconds, Package Lab stops by itself.

This time the banner says **Warning**, with “Got as far as “Committed”, but didn't reach “Finished”.” And there are two findings you didn't get before:

- **The content never called LMSFinish** (a warning);
- **The content never set exit** (a note).

Nothing is wrong with the course. Nobody left it, so it never had the chance to finish. Both findings say as much in their explanation.

This is the first lesson of reading a run: **a finding describes what happened in this run, not a fault in the course.** A timed run that nobody used will always stop before *Finished*. Before you report a fault, ask whether the run gave the course a fair chance to do the thing it's accused of not doing.

Step 6: look at the runs

Runs, at the top of the sidebar, now has two runs. Click between them: every view changes to show the run you've chosen. To see what the files alone say again, open the package again (**File › Open Recent**).

Every run is saved, so you can open it again later with **File › Open Saved Run...**.

Try it

1. Run `01-works.zip` with **I stop it**, go to page 3, choose **Save and exit**, and stop. Which milestones did it reach? Is there a note about exit? (There shouldn't be: **Save and exit** sets exit to `suspend`.)
2. Run it again with **10 s** and compare the two runs' banners. Which findings appear in one and not the other, and why?
3. Open the **API** view on your first run and find the call that set `cmi.core.score.raw`. What score did you get?

Check yourself

1. What does the banner's word tell you, and what does its sentence tell you?

2. A timed run of a course says “The content never called LMSFinish”. Is the course broken?
3. Which milestone would you look at first for a course that shows a blank page?
4. How do you go back to looking at what the package's files alone say?

Next: Reading the evidence.

6. Reading the evidence

What you'll learn: what each of the main views records, what normal looks like in each, and how to move from a finding to the evidence behind it. Keep a run of `01-works.zip` from chapter 5 selected while you read: the one where you chose **Save and exit**.

Findings are Package Lab's conclusions. The views are its evidence. An analyst reads both: the findings to know where to look, the views to check the findings are right and to see what they don't say.

The API view: the conversation with the LMS

Every SCORM call the content made, in order, with its element, value, result and error code, and below them the values the LMS held at the end.

On a good run of `01-works.zip` it reads like the healthy session from chapter 2. Here are its first calls, shortened (your view also shows reads of `cmi.launch_data` and a second read of the status):

Call	Element	Value	Result
`LMSInitialize`			`true`
`LMSGetValue`	`cmi.core.student_name`		`Learner, Test`
`LMSGetValue`	`cmi.core.entry`		`ab-initio`
`LMSGetValue`	`cmi.core.lesson_status`		`not attempted`
`LMSSetValue`	`cmi.core.lesson_status`	`incomplete`	`true`
`LMSGetValue`	`cmi.suspend_data`		
`LMSSetValue`	`cmi.core.lesson_location`	`1`	`true`
`LMSCommit`			`true`
…			

What to look for:

- **A `false` result**, and the error code beside it. An error on a call the course thinks worked is a silent failure. The commonest codes are in the table below.
- **The order.** Reading `cmi.core.entry` and then never reading the location is a course that won't resume.

- **What was never called.** No `LMSFinish`? No `LMSSetValue` of the status? Absence is evidence too.
- **The stored values at the end**: what an LMS would keep.

The error codes you'll see most:

Meaning	SCORM 1.2	SCORM 2004
Called before the session started	301	122 (get), 132 (set), 142 (commit)
No such element (often a typo)	201 or 401	401
The element is read-only	403	404
The value is the wrong type or too long	405	406

Lookups: did the content find the LMS?

Every time the content read `window.API` or `window.API_1484_11`, in any frame, with whether it found one and the line of code that looked. It also shows each frame's depth and opener, and any time the content *assigned* those names itself.

On a good run there are a few lookups, the last one found. This view answers the most common question about a course that doesn't track: did it look at all, did it look in the right place, and did it look for the right name?

Network: every request

Every file and data request: its status, type, and the code that made it. A good run has only `200` responses from the package. Look for:

- **404 (not found)**: a missing file, or the wrong capitals;
- **requests to the server's root** (`/api/…`) or **another site**: the content expects a platform that isn't here;
- **requests with no response** at the end of the run: something the content is still waiting for.

Errors and Console

Errors lists, in time order, uncaught errors (the ones that stop a script), caught errors (ones the content caught itself, when **Record caught errors** is on) and browser warnings, each with where it happened. **The first error is usually the one that matters**: later ones are often knock-on effects.

Caught errors matter because content often catches its own failures and carries on quietly. A course that catches the error from a failed LMS lookup and switches to “offline mode” shows no uncaught error at all.

Console is whatever the content printed with `console.log`. Authors often leave useful messages there.

Boot path: how the course should load, and how it did

Boot path is worked out from the files: every page, script, stylesheet, frame and data file the launch page loads, in loading order, indented under whatever loads it. Each row says:

- **How** it is loaded: a `tag` on the page, a `request` (fetch or XHR), a script that sets `src`, an `import`, and so on;
- **From**: the file and line that loads it;
- **In the files**: *in package*, *missing*, *wrong capitals*, *server path*, *outside package*, *author's computer* or *other site*.

With a run selected, it adds **In the run**: when each file was asked for, and its status, or *never asked for*. The first file marked *never asked for* is where loading stopped or went another way.

Below the loading path are **Script order** (the order scripts actually run, with `defer` and `async` taken into account) and **Paths built at run time**, which can't be checked from the files.

Timeline: everything at once

Timeline puts the run on one time axis: milestones, requests (bars, red if they failed), scripts, API calls (dots, coloured by kind), errors (red ticks) and what the screen showed. Use **Zoom** and **Start** to look closer, hover for details, and click to open the view an item came from.

The timeline answers “what happened just before it went wrong?”, which no single view can. A request that failed a moment before the content gave up on the LMS, or an error just after a script loaded, stands out here.

Frames and Scripts

- **Frames**: every frame navigation and its target, including frames on another origin, workers and popups.
- **Scripts**: every script the browser read, in order, with what loaded it. **Code** opens one.

Package and Files: the package itself

These two views are about the package, and look the same with or without a run.

- **Package**: the manifest's items and resources, and every match from a scan of the scripts: API names, searches, requests, server paths and **mode switches** (names such as `previewMode` or `standalone`, where content often decides whether it is in an LMS). Each line number opens the code.
- **Files**: the package's files as a tree, with an editor. Chapter 8 uses it.

The code view

Anything that names a place in a file opens the code there: a `file:line` in a finding's evidence (it's a link), **Code** in Scripts and Execution, a line number in Package, **View code** in Files. It shows:

- **Formatted** code, so a minified file is readable (or **Original**, as it is, or the **Source map**'s original source when the file has one);
- **Coverage**, with a run chosen: green lines ran (with how many times), red didn't, amber partly ran;
- **Search** and **Go** to a line.

Coverage is a quiet superpower. A red block in the middle of green tells you which way the code went, without your having to understand it all.

From a finding to the evidence

Every finding names its evidence. The skill is following it:

1. Read the finding's **evidence** lines.
2. Click the `file:line` to see the code.
3. Open the view the evidence came from (Network for a request, Errors for an error, Lookups for a lookup) and find the row.
4. Look at what happened *just before* it, in that view or the Timeline.

Try it

Use your run of `01-works.zip` from chapter 5 where you chose **Save and exit** at page 3, before the quiz. **Runs** lists runs in the order you made them, newest at the bottom.

1. In **API**, find the calls made when you chose **Save and exit**. Which element was set to `suspend`?
2. In **Lookups**, how many times did the content look for `API` before it found it? On which line of `scorm.js`?
3. In **Boot path**, which file is loaded with `defer`, and what does **Script order** say about when it runs?
4. In the code view of `lesson.js`, with your run chosen, find a line coverage shows in red. Why didn't it run? (Hint: did you get every question right?)
5. In **Timeline**, use a run where you submitted the quiz, and hover over the API dots around that time. What was written?
6. In **Frames**, how many frames does the run have, and which one is the course in? Compare with what chapter 2 said about where an LMS puts a course.

Check yourself

1. Which view tells you whether the content looked for the LMS, and where?
2. A run has five errors. Which do you read first?
3. What does *never asked for* in the Boot path's **In the run** column mean?
4. Why can a course with no uncaught errors still have failed because of an error?
5. What does a red line in the code view's coverage tell you?

Next: Your first diagnoses.

7. Your first diagnoses

What you'll learn: a routine for any broken package, and how to use it on four practice packages. By the end you should be able to say, for each one, what is wrong, how you know, and what would fix it.

The first-look routine

Do this with every package, real or practice, before anything clever.

1. **Open it and read the static findings.** Errors here (a misnamed manifest, a missing file, wrong capitals, a version mismatch) often explain everything downstream. Note them, but don't trust them blindly yet.
2. **Run it with the default settings for 10 or 20 seconds**: everything as it comes, except **Stop after** set to 10 s or 20 s (its own default is *I stop it*). Don't touch the course. Throughout this guide, "run it for 10 seconds" means this.
3. **Read the banner sentence.** Where did it stop? That milestone tells you which kind of problem you have (see the table below).
4. **Read the first finding that explains the stop**, and follow its evidence into the views (chapter 6).
5. **Write down your diagnosis in one sentence**: "It stops at *X* because *Y*, shown by *Z*." If you can't fill in *Z*, you haven't finished.

Where it stopped, and where to look:

Stopped before	The kind of problem	Look in
Launch page loaded	The manifest's launch path is wrong, or that file is missing	Package, Network
Scripts loaded	A file didn't load	Network, Boot path
Content shown	The page never drew anything	The blank-page or loading-screen finding, Errors, Boot path
Looked for the API	The LMS code never ran	Errors, Lookups, Package (mode switches)
Found the API	It looked in the wrong place, or for the wrong name	Lookups
Initialised	It found the LMS, but the first call failed	API (the call's result and error code)

Stopped before	The kind of problem	Look in
Read saved data, Wrote data or Committed	The conversation stopped part-way	API
Finished	Often nothing: nobody left the course	Use it yourself, with **I stop it**

The recipes are longer versions of this routine for six of the commonest symptoms. Use them on real courses.

Exercise 1: `02-blank-page.zip`

"The course is just a white page."

1. Open it. How many static findings are there, and of what severity?
2. Run it for 10 seconds. What does the banner say? Which milestone is the first cross?
3. Find the finding **The content shows a blank page**. Its evidence is a list, *in loading order*, of what went wrong on the way. What is first?
4. Open **Boot path**. Find the two scripts the launch page loads after `scorm.js`. What does **In the files** say for each? What does **In the run** say?
5. One of the problems is a file that is truly missing. The other is a file that *is* in the package. What's the difference, and which of the two matters more for this course? (Hint: look at what the launch page contains in **Files**, and the note **The launch page has no content of its own**.)

Write your diagnosis, then check it in the answers.

Exercise 2: `03-not-tracked.zip`

"I finished it, but the LMS says I never started."

1. Open it. Read the static finding. What does it claim?
2. Run it for 10 seconds. The course shows its pages, but read its first line of text in the course window (or in the evidence for **Content shown**). What does it say?
3. Which milestone is the first cross?
4. Open **Lookups**. What name did the content look for? What does the finding say the run *offered*?
5. The finding's **Try:** suggests a test with the **API** setting set to **Both**. You'll learn to run it in chapter 8; for now, write down what you expect it to show.

Check your diagnosis in the answers.

Exercise 3: `04-loading-forever.zip`

"It says Loading and never gets past it."

1. Open it and read the static findings. There's an error about a data file. Open the file in **Files** and look at it. Can you see what's wrong? (Look closely at the end.)
2. Run it for 10 seconds. The banner names a finding about a loading screen. Which element did Package Lab recognise as the loading indicator?
3. Open **Errors**. What is the first error, and which file and line does it name? Click through to the code.
4. There's a warning: **Code that loads the next part of the content never ran**. Which file was never loaded? Look at the code that would have loaded it. Why didn't it run?
5. Open **Timeline**. Where does the error sit relative to the request for the data file?

Check your diagnosis in the answers.

Exercise 4: `05-error-in-the-console.zip`

"The browser console shows an error. Does it matter?"

This one is different. Somebody saw an error and wants to know if it's the cause of their problems.

1. Open it. The static finding is a **Problem**. What does it say will happen?
2. Run it for 10 seconds. How far did the run get? Compare it with a 10-second run of `01-works.zip`.
3. Open **Errors**. The error is real. Where is it?
4. Look at the API view and the stored values. Did the course track?
5. So: is the error a fault, and does it matter? What would you tell the person who asked?

Check your answer in the answers.

The lesson here is the most useful one in this chapter. **A static finding is a prediction. The run is the test of it.** Package Lab's static checks are cautious: they warn about what *can* go wrong, and sometimes it doesn't. Its severity is how bad the problem would be if it happened, not proof that it did. Always confirm with a run before you tell anyone their course is broken, and always say *how much* it matters.

What you've learnt

- Every diagnosis starts with the same routine: static findings, a short run, the first missed milestone.
- The milestone tells you which views to open.
- The first problem in loading order, or the first error in time order, is usually the cause. Later ones follow from it.
- Static findings predict; runs confirm.

Check yourself

1. A run's first cross is at **Looked for the API**. Which views do you open first?
2. Why does the blank-page finding list its evidence in loading order?
3. A package has a static Problem, but a run reaches every milestone you'd expect. What do you report?
4. Say the one-sentence diagnosis for any one of the four exercises, with its evidence.

Next: Testing a theory.

8. Testing a theory

What you'll learn: how to prove a diagnosis instead of guessing it, by changing one thing in Package Lab and running again. You'll use launch settings, changed files, pins, answers and experiments.

Why prove it?

In chapter 7 you formed diagnoses from findings. A diagnosis is a theory. Before you tell an author to republish, or an LMS administrator to change a setting, you want to know the change will work. Package Lab lets you make that change in its own copy of the world, without touching the course, and see whether the course gets further.

The rule, from chapter 1: **change one thing, run again, compare.** If you change two things and it works, you don't know which one mattered.

"Further" is measured by the milestones. A change that moves the banner from "Got as far as *Looked for the API*" to "Got as far as *Committed*" helped. One that makes no difference didn't.

Tool 1: launch settings

The quickest theories to test are about how the LMS hosts the course. The inspector's **Settings** hold them. The three you'll use most:

Setting	Where	What it tests
API	LMS	Which API objects the test LMS offers: *as the manifest says*, *both*, *1.2 only*, *2004 only* or *none*. If *both* works where *as the manifest says* didn't, the manifest and the scripts disagree about the version. *None* shows what the course does with no LMS at all
Frame depth	Advanced	How many empty frames sit between the LMS's window and the course (0 to 3). A course that only searches a few parents fails when it's deeper. Many LMSs put content two or three frames deep
In a new window	Advanced	The LMS opens the course with `window.open`, so the course has to find the API through its opener

After any change, glance at the **summary chip** beside **Run**: it lists every setting that isn't at its default. When you've finished testing, put them back. Most settings are remembered even after you quit Package Lab (chapter 4), so a setting you forget today surprises you tomorrow.

Try it: `03-not-tracked.zip`

In chapter 7 you found that the manifest declares SCORM 2004 while the scripts use 1.2. Prove it.

1. Open `03-not-tracked.zip` and run it for 10 seconds with the default settings. Note how far it got.
2. Set **API** (in **Settings › LMS**) to **Both**. The summary chip now shows it.
3. Run again for 10 seconds. How far does it get now? What does **Lookups** show?
4. Set **API** back to **As the manifest says**.

You have proved the cause: the content works when the LMS offers the API it expects. You've also shown the two possible fixes: republish the course for SCORM 2004, or correct the manifest to say 1.2. (Package Lab can make the second fix for you, in chapter 13.)

Tool 2: changing a file

Sometimes the theory is about the course's own code: “if this line weren't here, it would work”. Package Lab lets you edit any text file of the package and run the edited copy. The original is never changed.

1. Open **Files** and choose a file.
2. Edit it, then choose **Save as override**. It's marked **changed**.
3. **Run**. The run uses the changed file. Its findings start with “This run used 1 changed file”, and if an earlier run of the same item had no changes, a second note says whether the change helped: it got further, got less far, or made no difference.
4. **Show changes** shows your edit as a diff. **Revert** undoes one file, **Revert all** undoes them all.

Changes last for the session and go with every run, including experiments, until you revert them. A non-text file can be swapped with **Replace from a file...**.

Try it: `02-blank-page.zip`

1. Open `02-blank-page.zip`, open **Files**, and choose `lesson.html`.
2. Correct the capitals of the script path you identified in chapter 7. Choose **Save as override**.
3. Run for 10 seconds. What does the second note at the top of Findings say? Is the page still blank?
4. Read the blank-page finding again. Its first cause has changed. What is it now, and on which line? (Think about where in the page the script tag is, and what exists at that moment.)
5. Fix that too, with a second change to `lesson.html`, and run again. What does the page show now? Is the missing script still a problem?

This is what most real diagnoses look like. One fault hides the next. Fix the first cause, run again, and read the first cause again, until the course gets as far as it should. This one is an exception: even fixed, **Content shown** keeps its cross, because the sample page has too little text for that rule, and the page never talks to the LMS. Stop when the page shows its text with no error (the answer explains).

Tool 3: pinning a value

Courses often decide how to behave from a variable: `previewMode`, `config.lms`, `isStandalone`. A **pin** makes the content read the value you choose, from before its first line of code runs, whatever it sets itself.

1. Open the inspector's **Settings › Advanced** (or the **Lookups** view, which has the same list), and find **Pins**.
2. Type the name (`previewMode`, or a dotted name like `config.mode`), the value as JSON (`true`, `false`, `3`, `"text"` with quotes), and choose **Add pin**.
3. Choose **Run with these**, or **Run**.

Lookups then shows every read of a pinned value, and every time the content *tried to change* it, with the line. That line is often the code that decides the course's mode, which is exactly what you were looking for.

If the content declares the name with `let` or `const` at the top of a script, a pin can't replace it. The findings say so and point at the line; change it in **Files** instead.

Try it: `06-stuck-connecting.zip`

"It says Connecting to the LMS… for ever, and nothing is recorded."

1. Run it for 10 seconds and read the banner. Where did it stop? Is there an *uncaught* error in **Errors**?
2. With no error, the content chose not to start its LMS code. Open `lesson.js` in the code view with your run chosen, and look at the coverage near the end of the file. Which line ran, and which didn't? What does that line test?
3. Pin that name to `true`, and run again. How far does it get?
4. Remove the pin. Now prove it a second way: change the line in **Files** so it doesn't test anything, save it as an override, and run. Same result?

Check your answer in the answers.

Tool 4: answering a request

Content made for a hosting platform often asks that platform for something before it starts: a configuration file, a user record, a licence. In an LMS nobody answers, so it stops, or falls back to a mode that never talks to the LMS. An **answer** gives the content a response to a request, so you can see what it does next.

1. In **Network**, find the failed request, or the one to the server's root or another site, and choose **Answer this request...**.
2. Check the pattern (a path, a glob with `*`, or a `/regex/`), the status, the content type and the body. The body starts as `{}`, an empty JSON object; put real values in if the course needs them.
3. Choose **Add and run with these**. Answered requests are marked **answered** in Network.

The list is in **Settings › Advanced › Answers**. The findings name the requests that were answered, and any answer nothing asked for.

Try it: `07-works-on-the-platform.zip`

"It worked in the authoring tool's preview, but not in our LMS."

1. Run it for 10 seconds. It looks as though it got quite far. Which milestones did it reach? Now read the first finding. Who was the content really talking to?
2. Open **Network**. Which request failed? What kind of path is it?
3. Answer it with the default body `{}`, and run. The banner may say the same milestone as before. Read the findings again: which one has gone? Open **Lookups** and **API**: whose `LMSInitialize` was called this time?
4. What does that tell you about the course, and what would you ask its author?

The milestone didn't move, but the course is talking to a different LMS. Milestones measure how far a run got, not who it got there with. Always read the findings, not only the banner.

Check your answer in the answers. Chapter 12 shows how Package Lab can work out what a real answer should contain.

Tool 5: experiments

Changing one setting at a time is the method, but when you don't know which setting to try, Package Lab can try them all for you. Open **Experiments**, choose a set of variants, and **Run all**. It runs the item with your current settings, then once with each variant, and fills a table as each finishes: how far it got, API calls, errors, the verdict, and how it compares with the first run.

Variants	Tries
api	The API as the manifest says, both, 1.2, 2004 and none
frames	Frame depth 0, 1, 2 and 3, and a new window
standard	Both of those
lms	Every LMS profile (chapter 11)
storage	Storage normal, then blocked
browsers	Chromium, then WebKit

If a variant got further than the first run, a finding says so and names the settings that changed. Click a row to open that run. Each run lasts as long as **Stop after** (10 s if it says *I stop it*).

Try it

1. Run the **api** experiment on `03-not-tracked.zip`. Which variants get further than the default? Does it agree with what you proved by hand?
2. Run the **standard** experiment on `01-works.zip`. Which variants get less far than the default, and why would you expect the API ones to? Which frame variants make no difference, and what does that tell you about how far up its `scorm.js` searches?

Keeping track

As soon as you're testing theories, you'll have a dozen runs and forget which was which. Two habits help:

- **Read the run's label.** It names the profile, browser, device, storage mode and so on when they aren't the defaults.
- **Keep a log.** One line per run: what you changed, how far it got. It's the evidence you'll hand on in chapter 13.

Check yourself

1. Why change one thing at a time?
2. What's the difference between a pin and a changed file? When would you use each?
3. A course gets further with **API: Both**. What two fixes does that suggest?
4. You answered a request and the course got further. Is the course now fixed?
5. When would you run an experiment rather than change a setting yourself?

Next: Macros: playing the learner.

9. Macros: playing the learner

What you'll learn: how to make Package Lab click through a course for you, the same way every time. You'll record a macro, read and edit it, add waits so it doesn't click too soon, see how clicks follow their button when the layout changes, and read and write macro files.

Why macros matter

Some faults only show after a learner has done something: moved a few pages, answered a question, left part-way. You can do that by hand once. But an analyst runs a course ten or twenty times while testing theories, and needs every run to do the same thing, or the runs can't be compared. A **macro** is a list of clicks, keys, typed text, scrolls and waits that Package Lab plays into the course during a run. Every run plays it identically.

Macros are needed for the save-and-resume test (chapter 10), and they're useful for any fault that only appears on page 5.

Four ways to make one

Way	Where	Best for
Record	The toolbar, while a run goes (⌥⌘R)	The usual way
Record clicks	The **Live** view	Seeing exactly where each click lands; the web interface
Use this run's clicks	The inspector's **Macro** segment, after a run	You used the course and only afterwards wished you'd recorded it
A file	**Import…** in the Macro segment, or `--macro` on the command line	Macros written or edited by hand, shared, or kept with a project

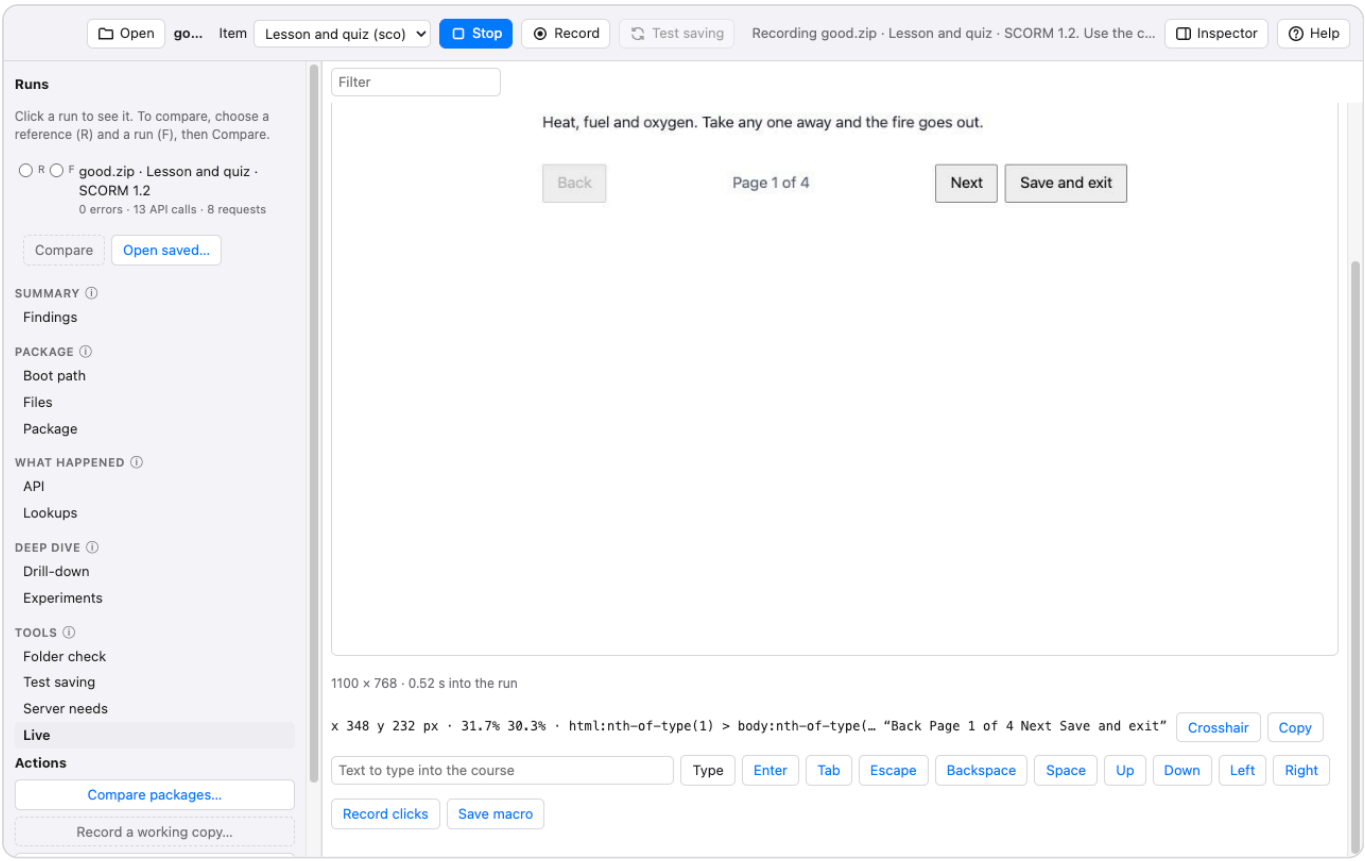
Recording with Record

1. Set **Stop after** to **I stop it**, and **Run**.
2. Choose **Record** in the toolbar. It reads **Recording…**.
3. Click through the course as a learner would, in the course's own window (not Package Lab's).
4. Choose **Record** again, or **Stop**. Package Lab asks “Save 6 steps as the macro?”. **Save** replaces the macro you had with the new recording; **Discard** throws the recording away and keeps your previous macro.

Only clicks made after you pressed **Record** are kept.

Recording in the Live view

Live (in the sidebar's **Tools** group; or **Show in Package Lab** in the toolbar while a run goes) shows the course window inside Package Lab. Click the picture and the click goes to the course; with the picture focused, keys go to it too.



Under the picture, the **coordinates bar** follows the pointer:

```
x 412  y 233 px · 37.5% 29.1% · button#next “Next”      [Crosshair] [Copy]
```

The pixels are in the course window's own size; the percentages are the same point as fractions across the window, which is what a macro stores; the last part is the element under the pointer. **Crosshair** draws guide lines, and **Copy** (⌘C) copies the position and the element's selector, handy when you're writing a macro by hand.

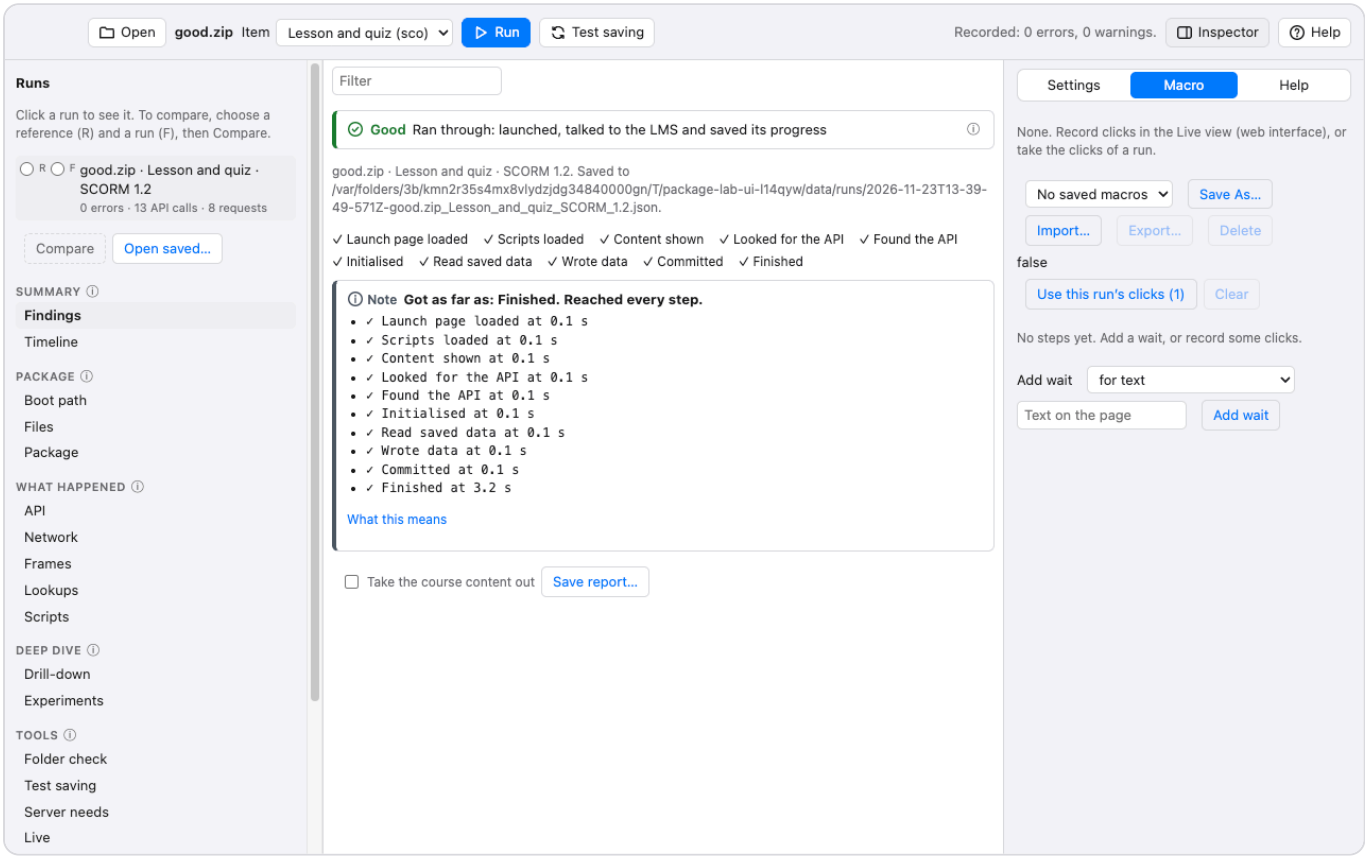
Choose **Record clicks**, use the course through the picture, then **Save macro**. This is also how you record in the web interface, from an iPad or another computer.

Reading a macro

The inspector's **Macro** segment shows the macro as a table:

#	When	Action	Where
1	+0.5 s	Click	165, 80 `#next “Next”`
2	+0.4 s	Click	165, 80 `#next “Next”`

- **When** is the time since the step before, or, for a wait, what it waits for.
- **Where** is the position, in pixels at the size the macro was recorded at, and the element the click was made on.
- Above the table: how many events, over how long, “recorded at 1100×800”.



What a macro remembers

The element, as well as the position. A click is saved with the element it hit: a selector (`#next`), its role and up to 60 characters of its text. When the macro plays, the click goes to that element *wherever it is now*, if exactly one element matches and can be seen. Only if it can't be found does the click fall back to its recorded position, and the run gets the note “2 clicks didn't find their element and used their recorded position”.

The window size. Positions are fractions across the window, so a macro remembers the size it was recorded at, and a run with no **Device size** of its own plays at that size. If you choose a different size, yours wins, and the run notes “The macro was recorded at 1100×800 and played at 390×844: clicks may miss”.

Typed text, only if you asked. Text typed in the course is kept only if **Keep typed text in runs (for macros)** (**Settings › Recording**) was ticked, and never from a password field or a card or one-time-code field. Text typed through the Live view while **Record clicks** is on *is* kept, so don't type secrets there.

Not pointer moves. Recording drops them. It stops at 2,000 events or 100 KB of text.

Waits: the cure for clicking too soon

Timing is the commonest reason a macro misses. A button that appears after a slow request, an animation or a three-second intro isn't there when the click comes. A **wait** step blocks the macro until something has happened:

Wait for	Means
Text	Some text is visible on the page, in any of the course's frames. Text that is there but hidden doesn't count
A milestone	The run reaches a step, such as *Initialised*
Quiet network	No request has been pending for so long
A time	Some milliseconds, no matter what

A text, milestone or quiet wait gives up after its timeout (30 s unless you set another). Then the macro stops there and the rest of its steps are skipped. The run says “The macro stopped at step 1: waited 30 s for “Next” and it never appeared”, and goes on recording until **Stop after** ends it.

Add wait puts a wait after the chosen row (at the end if none is chosen). Use ↑ to move it earlier.

Prefer waiting for *something* (text, a milestone) over waiting for a *time*. A time that works on your Mac is too short on a slow LMS.

Editing

Choose a row (click it, or Tab to it and press Enter) to change it in the form below the table. ↑ and ↓ move a step, ✕ deletes it. Changing a step's delay moves every later step by the same amount, so the rest keeps its timing.

While a run is going, **Play step**, **Play from here** and **Play to here** play part of the macro into the course, so you can try a change without a whole run.

Keeping macros

The current macro plays at the start of every run until you **Clear** it or close the app. That's easy to forget. If runs start clicking things you didn't expect, look at the **Macro** segment.

To keep one:

- **Save As...** names it and keeps it for this package (in the data folder's `macros/`). The menu at the top of the segment lists a package's saved macros; choose one to load it.
- **Export...** writes it to a file; **Import...** reads one. Neither changes your saved macros until you **Save As...**.

Macros need Chromium. With **Browser: WebKit**, the run says the macro wasn't played.

Macro files

A macro file is JSON. You'll rarely write one from scratch, but reading one helps, and editing one is sometimes quicker than re-recording:

```
{ "format": "package-lab-macro", "version": 2, "viewport": { "width": 1100,
"height": 800 }, "events": [
  { "t": 400, "type": "wait", "text": "Next", "timeout": 10000 },
  { "t": 500, "type": "click", "x": 0.15, "y": 0.1,
    "target": { "sel": "#next", "text": "Next", "role": "button", "frame": [0] } }
] }
```

- `t` is milliseconds since the run started; the gap between two events is the pause between them.
- `x` and `y` run from 0 to 1 across the window recorded in `viewport`.
- `target` ties a click to its element: `sel` is a CSS selector, and `frame` the frames to go down from the top window. Edit `sel` to point a click at another element.
- Other types are `key` (`Enter`, `Tab`, arrows...), `text`, `wheel` and `move`.

To print a macro file as the editor's table *(command line, chapter 14)*:

```
package-lab macro show macro.json
```

The full format is in the command line reference.

Try it: record one

1. Open `01-works.zip`. Run it with **I stop it**, choose **Record**, choose **Next** twice, then **Record** again and **Save**. Stop the run.
2. Read the **Macro** segment. How many steps? What element is each click tied to? What size was it recorded at?
3. **Run** again with **10 s**, and don't touch anything. In **API**, what was the last `cmi.core.lesson_location` written?
4. **Save As...** `two-pages`. You'll use it in the next chapter.

Try it: `14-next-button-late.zip`

"Our automated test clicks Next and nothing happens."

The practice kit includes a macro for this package, `14-next-button-late.macro.json`.

1. Open `14-next-button-late.zip`, then choose `14-next-button-late.macro.json` here to load it into the **Macro** segment (or **Import...** it from the practice folder). Read it: when do the clicks come?
2. Run it for 10 seconds. In **API**, how far did the lesson location get? Did the clicks do anything?
3. Use **Live**, or run with **I stop it** and watch the course window. When does **Next** appear?
4. Choose **Add wait**, set it to wait for the text `Next`, and move it to the top with ↑. Run again. Now how far does the location get?

Check your answer in the answers.

Try it: `15-button-moves.zip`

"The macro works on my screen but not on the test machine."

1. Open `15-button-moves.zip` and load `15-button-moves.macro.json`. Run for 10 seconds. Does it move pages?
2. Set **Device size** to **iPhone** and run again. Read the note about the size. Did the clicks still work? Why?
3. Now break the tie: **Export...** the macro, open the file in a text editor, change both `"#next"` to `"#nope"`, save it and **Import...** it again. Run at **iPhone** size. Did it still work? Read the notes.
4. With the `#nope` macro still loaded, set **Device size** to **Custom size...** 2400 × 800, and run again. What happened this time? Where is **Next** at that width, compared with where the macro's position says?

5. Put **Device size** back to **Window as it is**, and **Clear** the macro.

Check your answer in the answers.

Check yourself

1. Why does an analyst want a macro, even for a course they could click through by hand?
2. A click's element can't be found. What does Package Lab do, and how do you know it happened?
3. Your macro works at 1100 × 800 but not on an iPad size. Name two possible reasons.
4. Why is “wait for text *Next*” better than “wait 3 seconds”?
5. Every run since lunch has been clicking through the course by itself. Why?

Next: Saving and resuming.

10. Saving and resuming

What you'll learn: how to check that a course saves the learner's place and picks it up again, by hand and with Package Lab's save-and-resume test. You'll need the macro you recorded in chapter 9.

“I have to start again every time” is one of the commonest complaints about e-learning, and one of the hardest to reproduce, because it needs two sessions and a learner who left part-way. Package Lab can be that learner.

What a working resume needs

From chapter 2, a course resumes properly when:

1. as the learner moves, it **writes** the bookmark (`lesson_location`) and its own state (`suspend_data`);
2. when the learner leaves early, it sets **exit** to `suspend`, and ends the session (`LMSFinish` or `Terminate`);
3. next time, the LMS **offers a resume**: `entry` is `resume`, and the saved values are given back;
4. the course **reads** the bookmark and suspend data, and **uses** them to go back to where the learner was;
5. when it saves again, it **adds to** what was there rather than starting over.

A fault at any step looks the same to the learner: they start again. The job is to find which step broke.

Testing it by hand

You can do the whole thing yourself:

1. Set **Stop after** to **I stop it**, and **Run**.
2. Go a few pages in, then leave the course the normal way (its exit or close button).
3. **Stop**. In **API**, check that the location and suspend data were set, exit was set to `suspend`, there was a commit, and `LMSFinish` was called.
4. In **Settings › LMS**, tick **Resume last run's data**, and **Run** again. The data model starts with what the last run stored.
5. Did the course go back to where you left? In this run's **API** view, did it read `cmi.core.entry`, `lesson_location` and `suspend_data`, and then write the same location back?

That works, but it's slow, and it's easy to forget a step. The save-and-resume test does it for you.

Test saving

Test saving in the toolbar (⇧⌘R) runs the course, plays a macro of the learner's clicks, and then leaves the way a learner closing the window does: the course is sent to a blank page, so its `beforeunload`, `pagehide` and `unload` handlers run, and the window is closed. Package Lab keeps what the LMS held. Then it runs the course again with that data, as a resume.

The **Test saving** view shows the two runs side by side:

Row	Run 1: leaves	Run 2: resumes
Entry offered	`ab-initio`	`resume`
Location saved (first, last)	`1, 3`	`3, 3`
Exit set	`suspend`	
Session ended by the content	yes, while leaving	no

and the findings, which stop at the first thing that went wrong: **Leaving the course saved nothing**, **...didn't end the session**, **The LMS started a new attempt instead of resuming**, **On resume, the content didn't read its saved place**, **The content read its saved place but started from the beginning**, and so on. The last one you want to see is **Saving and resuming worked**.

Each run lasts as long as **Stop after** (10 s if it says *I stop it*). Both runs are in **Runs**, so you can open their API views. Test saving runs in Chromium only.

Why it needs a macro

Without a macro, Test saving checks only what the course saves by itself at start-up. Most courses save as the learner moves, so for a real test something must move: that's the macro from chapter 9. Run 1 plays it; run 2 plays nothing, so you can see where the course puts the learner by itself.

The macro carries over. The current macro isn't tied to a package: it stays loaded when you open another one, and plays in every run until you **Clear** it. That's handy here, because every practice package has the same **Next** button. It's a nuisance later, so the chapter ends by clearing it.

Try it: `01-works.zip` first

1. Open `01-works.zip`. In the **Macro** segment, choose your saved macro `two-pages` from the menu at the top (or record one again: **Next** twice).
2. Choose **Test saving**. Read the table. What location did run 1 save last? What did run 2 read?
3. The last finding should be **Saving and resuming worked**. This is what good looks like.

Try it: `08-forgets-its-place.zip`

"When I go back, I'm at the start again."

1. Open it. The macro from `01-works.zip` is still loaded: check the **Macro** segment.
2. **Test saving**. Read the row **Entry offered** for run 2, and the row **Exit set** for run 1.
3. Read the finding. Which of the five steps above broke?
4. Check it by hand: in run 1's **API** view, is there a `LMSSetValue` of `cmi.core.exit`?

Check your answer in the answers.

Try it: `09-starts-over.zip`

"It asks me where I was, then starts from page 1 anyway."

1. With your macro, **Test saving**.
2. Compare **Location saved (first, last)** for the two runs. Where did run 1 leave? Where did run 2 start?
3. Read the finding. This time, which step broke? Did the course read its bookmark?
4. In run 2's code view of `lesson.js`, find where the location is read and what is done with it.

Check your answer in the answers.

Two warnings about this test

- **It uses Package Lab's test LMS's rules.** It offers a resume only after exit is `suspend`, as many LMSs do. Some LMSs are more forgiving, some stricter. Confirm the behaviour in the real LMS.
- **Suspend data has limits.** A finding "Suspend data is *N* characters" means an LMS may cut it off: 4,096 in SCORM 1.2, at least 64,000 in 2004, or a profile's own limit. A course that resumes in Package Lab but not in the LMS may be over it.

Before you go on

Choose **Clear** in the **Macro** segment. The next chapters' runs should play no macro.

Check yourself

1. Name the five things a course must do for resume to work.

2. Why does Test saving need a macro for most courses?
3. A course resumes in Package Lab but not in the LMS. Name two things you'd check.
4. Why does run 2 of the test play no macro?

Next: Browsers, devices and LMSs.

11. Browsers, devices and LMSs

What you'll learn: why a course that works in one place fails in another, and how to reproduce Safari, iPads, blocked storage and the way particular LMSs host courses, without leaving Package Lab.

“It works for me” is usually true. The course works in the author's Chrome, on their laptop, in the authoring tool's preview. It fails for a learner on an iPad, in an LMS that serves content from another domain. The difference is the environment, and Package Lab can imitate each part of it.

The browser: Chromium or WebKit

Settings › Browser and device › Browser chooses the engine:

- **Chromium**: Package Lab's own browser, the engine inside Chrome and Edge. Everything works.
- **WebKit (Safari's engine)**: the engine of Safari, and of every browser on an iPhone and iPad. It needs Package Lab run from its source code, with WebKit added (chapter 14): the downloaded app can't run WebKit, and says so if you choose it.

A WebKit run uses the same test LMS, files, changes, pins, answers, profile and storage setting. It records requests, console messages, uncaught errors and API calls, but not coverage, caught errors or script sources, so it can't be traced and the code view is empty. Its label says `webkit`.

The commonest cause of a WebKit-only failure is code that uses something only Chrome has, such as `navigator.userAgentData`. The error names it, with the line.

The device: phone and tablet sizes

Device size runs the course at **iPhone** (390 × 844), **iPad** (820 × 1180), **Android phone** (412 × 915) or a custom size. The presets also turn on touch, tell the course it's on a mobile device (its user agent included) and honour its `<meta name="viewport">`. A custom size changes only the size.

Use it for layout problems (a **Next** button off the screen, a course that refuses to run on a phone) and together with WebKit for “fails on my iPad”.

Storage: what the browser lets the course keep

Storage sets what the course's frames get:

Storage	What happens
Normal	Storage as usual. It lasts between runs while the app is open, shared by every package
Blocked (Safari-like)	`localStorage` and `sessionStorage` throw a `SecurityError`, cookies read as empty, IndexedDB fails. The strictest case: Chrome with third-party cookies blocked, and some Safari versions and settings, for a page inside another site's frame
Partitioned (empty)	Nothing refused, but emptied before the run: a learner's first visit, and how current Safari usually treats a frame from another site

If the course used blocked storage, the finding **The content needs storage the browser can block** lists every use, with its line. The fix is to keep the course's state in the LMS (`cmi.suspend_data`) and wrap browser storage in `try/catch`.

The LMS: profiles

Every LMS hosts a course a little differently. **Settings › LMS › Profile** imitates one:

- **generic** is Package Lab's own test LMS, the default;
- **moodle**, **scorm-cloud**, **cornerstone**, **successfactors**, **docebo**, **totara**, **blackboard** and **canvas** are approximations, from each LMS's public documentation and community reports, of what it does by default. **canvas** is Canvas's SCORM tool showing a package in a frame inside the course page, its default. The notes under the list say which documents, and where they were silent.

Choose a profile and its fields fill in. The ones that break courses:

Field	What it imitates	What breaks
Content origin	Course files served from the LMS's own domain (**same**) or another (**other**)	Across origins the course can't reach the LMS's windows, so it can't find the API
Frames, **Frame kind**	How many LMS frames wrap the course, and whether they're `iframe`s or an old `frameset`	Courses that search too few parents
Sandbox, **Allow**	The sandbox on the course's frame	A sandbox without `allow-same-origin` cuts the course off
Launch, **Popup delay**	In a frame, or a new window after a delay	Courses that only search parents, or look only once
API delay	The API appears a moment after launch	Courses that look once, at once

Field	What it imitates	What breaks
API on	The API on the top window, the parent, or the opener	Courses that look in only one place
CSP, **Headers**	Security headers on every file	Inline scripts, `eval`, other sites
Commit, **Slow commit**	Commits that fail or are slow	Courses that ignore a failed save
Suspend data	The LMS's own limit	Courses that save a lot
Cookies	Third-party cookies blocked	The same as blocked storage

Change any field to try a variation; choosing another profile clears your changes. An LMS's administrator can change most of these, so **treat a preset as a likely setup, not a fact**. When you know the real settings, set them here for each run, or write them down as your own profile file *(command line, chapter 14)*.

Experiments for this chapter

Three experiment presets, from the **Experiments** view, try all of it at once:

- **browsers**: Chromium, then WebKit;
- **storage**: normal, then blocked;
- **lms**: every profile in turn.

Try it: `10-fails-in-safari.zip`

"Fine on my PC, broken on my iPad."

1. Run it for 10 seconds with the default settings. How far does it get?
2. Set **Browser** to **WebKit** (it needs the source code with WebKit added: see chapter 14) and **Device size** to **iPad**, and run again. How far now? What does the course show? Would WebKit alone, at the normal size, have been enough?
3. Read **Errors**. What is the error, and which line of which file? What does that code use?
4. Now run the **browsers** experiment. Does the table agree?
5. Put **Browser** and **Device size** back.

With the downloaded app you can do step 1 only: WebKit needs the source code. Read the other steps, then the answer, which gives what each WebKit run shows.

Try it: `11-fails-in-some-lmss.zip`

"Works in one LMS, stuck on Connecting… in another."

1. Run it with the defaults. Does it work?
2. Run the **lms** experiment. Which profile gets less far than the others?
3. Click that row to open its run. Read the findings. What does the course need that this LMS doesn't allow?
4. Prove the cause with one setting, without a profile: set the profile back to **generic**, set **Storage** to **Blocked**, and run. Same result?
5. Find the line of code in the finding's evidence. What would you tell the author to change?
6. Put **Storage** back to **Normal**.

Check your answer in the answers.

Notice how step 4 narrowed it down. The profile changed several things at once. Setting one of them on its own showed which one mattered. That is the one-change rule again.

Try it: `18-lost-in-some-lmss.zip`

"Fine in our test LMS, never connects in Moodle or Canvas."

1. Run it with the defaults. Does it work?
2. Run the **lms** experiment. Which profiles get less far? Click one and read its findings.
3. One of the failing rows, **scorm-cloud**, also blocks storage. Read its finding: does Package Lab blame storage? Why not?
4. Find the one setting the failing profiles share. Test it on its own: set the profile back to **generic**, change only that setting to the failing profiles' value, and run. Then run the **frames** experiment.
5. Open **Lookups** for a failing run. How many windows did the course look at before it gave up? In `scorm.js`, find the line that decides how far it searches.
6. Put **Frame depth** back to 0.

Check your answer in the answers.

Try it: `19-forgets-its-answers.zip`

"It remembers my page, but not my answers."

1. Run it for 10 seconds. What's the first Problem? What does the **API** view show for the call it names?
2. Which limit is it over, and is it this LMS's limit or the standard's? Where in a profile could you change the limit to match a particular LMS?
3. Suggest two fixes, one for the author and one for the package's export.

Check your answer in the answers.

Try it: two LMS behaviours on `01-works.zip`

Profiles can break a course that has nothing wrong with it as written. Two common ones:

1. **The API comes late.** In **Settings › LMS**, set **API delay (ms)** to 1000, and run `01-works.zip` for 10 seconds. How far does it get? What does the course's first line say? `scorm.js` looks for the API once, at start-up. What would a sturdier course do?
2. **The save fails.** Set **API delay** back to 0, set **Commit** to **fail**, and run again. Which milestone is missed now? Does the course tell the learner? What would a learner lose in a real LMS?
3. Choose the **generic** profile again to clear your changes.

Check your answers in the answers.

Check yourself

1. A course fails only in WebKit. What's the first view you read, and what are you looking for?
2. Why can't you trace a WebKit run, and how do you get round it?
3. What's the difference between **Blocked** and **Partitioned** storage?
4. A course fails with the **scorm-cloud** profile, which changes both storage and frame depth. How do you find out which of the two matters?
5. Why should you treat a profile as an approximation?

Next: Going deeper.

12. Going deeper

What you'll learn: what to do when the findings don't name a cause: when nothing failed, nothing threw, and the course simply stopped. You'll use *why it stalled*, tracing, drill-down, server needs and a comparison with a working copy.

The findings so far have explained failures: a missing file, an error, a lookup that found nothing. The hardest courses don't fail. They wait, or choose a path, and nothing in the record says why. These tools find out.

Why it stalled

When a run stops short of **Wrote data** and nothing failed outright, the course is usually **waiting** for something. Package Lab records, in every frame, the timers, `postMessage` calls and `message` listeners the course sets up, and the finding **The content was waiting for: …** lists what was still open at the end:

- **Messages with no answer**: a message the course sent to another window (the LMS, a parent frame) that nobody answered. Its type is named: `{ type: 'player-ready' }` shows as “player-ready”.
- **Pollers**: a timer still firing at the end, every so many milliseconds, checking for something.
- **Messages that never came**: a listener in the course that never ran.
- **Requests** still open.
- **The last code that ran.**

Then ask: in a real LMS or player, what answers each of these? And give the course the same, with a pin, an answer or a changed file.

Try it: `12-waits-forever.zip`

“No errors anywhere, but nothing is recorded.”

1. Open it. Are there any static findings?
2. Run it for 10 seconds. Where did it stop? Is there an uncaught error in **Errors**, or a failed request in **Network**?
3. Read **The content was waiting for: …**. What did the course send, to whom, and what is it waiting to receive?
4. Open the lines its evidence names. Which line starts the LMS session, and when would it run?
5. How would you test your theory? (A pin can't send a message. What can change the course's behaviour?)

Check your answer in the answers. This is the hardest kind of fault: the files look fine, nothing fails, and nothing throws. Only the record of what the course was still waiting for gives it away.

Tracing the code

Findings say *what* the course did. To see *why*, turn on **Trace the code** in **Settings › Recording** and run again. Package Lab then records every function of the course's scripts that ran, and which way each decision went, with the values it tested. The course runs a little slower, but behaves the same.

The **Execution** view shows the last 500 events before the run went quiet, newest last:

- `startCourse() called` and `startCourse() returned`. A start with no end means it threw, or was still waiting.
- `if (cfg.lms) → false, cfg.lms = undefined`: a decision, which way it went, and why.
- `(the same 4000 more times)`: a loop, counted.

Read it from the bottom up. The last decisions are usually where the course gave up. **Code** opens the line, and the code view adds trace markers: `if: true ×1, false ×8`.

Tracing covers the package's own scripts (up to 2 MB each), not code from other sites, `eval`, or HTML `onclick` attributes. It keeps the newest 200,000 events. And it needs Chromium.

Drill-down

Drill-down does the reading for you. Choose a run, open **Drill-down**, and choose **Drill down**. If the run wasn't traced, Package Lab runs it again with tracing; if it needs to know where a value came from, it runs it once more, watching that value. Then it writes the answer as a numbered chain, each step the reason for the one before:

1. In `boot.js:3`, `if (window.player && player.ready)` was false because `window.player` was undefined. Otherwise it would have run `start()`, which never ran.
2. `player` was never assigned (watched). It is set at `js/player.js:1`, which is in a file that never loaded.
3. …

When the chain ends at something Package Lab can name (a failed request, a script that never loaded, a JSON file without the field, an API version the LMS didn't offer), the finding is an error. Otherwise it's a warning, and the last step says where to look.

Try it: `06-stuck-connecting.zip` again

You solved this one in chapter 8 with coverage and a pin. Now let drill-down solve it.

1. Check the **Macro** segment is empty (**Clear** it if not). Run it for 10 seconds, then open **Drill-down** and choose **Drill down**. Watch the runs it adds to **Runs**.
2. Read the chain. Does step 1 name the same line you found?
3. What does step 2 say about where the value should come from? Why is that a useful thing to tell an author?

Try it: `13-keeps-looking.zip`

"It never connects, but the vendor says it's fine."

1. Run it for 10 seconds. Where does it stop? Read **The content was waiting for**: what is it doing?
2. Look at the **Try:** of **The content looked for API and found nothing**. It suggests trying both APIs. Try it. Does it help? Why not?
3. Open **Lookups**. On which window does the course look, every time?
4. Drill down. What does step 2 say about where the API lives?
5. Prove it: in **Files**, change the one line in `boot.js` so the course looks where the API really is. Save it as an override and run. How far does it get?

Check your answer in the answers. Step 2 is a lesson in itself: a finding's **Try:** is the most likely fix for *most* courses that show it. Your job is to check whether this course is most courses.

Server needs

Courses made for a hosting platform ask it for settings before they start. You answered such a request by hand in chapter 8, with `{}`. But often the course needs specific fields, and you don't know which.

The **Server needs** view lists the run's requests to the server's root or to other sites. **Discover**, on one request, or **Discover all server requests**, works out what each one needs:

1. It runs the course without an answer.
2. It answers with a stand-in and watches every field the course reads from it, and how it uses each one.
3. It runs again with those fields filled in (`""`, `false`, `0`, `[]`), and again, until a round finds no new field (5 rounds at most; the command line's `--rounds` changes that).

The result lists, per request, the code that made it, every field read with its type and where it was used, and a stand-in response. **Save as response** adds it to your answers; edit the values first if the course needs real ones. The **Rounds** table shows how far each round got.

Try **Discover all server requests** on `07-works-on-the-platform.zip`. It reports that the course didn't read anything from `/api/player/config`: this course only cares that the request *succeeded*. That's a finding too: any `200` response would do, which is what your `{}` showed.

Comparing with a working copy

If the course works somewhere (the authoring tool's preview, another LMS, SCORM Cloud), record it there and compare.

1. **File › Record a Working Copy...**, enter the course's address, and choose **Record**. The course opens in Package Lab's browser, where you can sign in. Sign-ins are kept for the next recording, in Package Lab's own browser, separate from your usual one. Use the course the same way as before, then **Stop**. Package Lab watches every page in its browser, so the course's calls to the real LMS's API are recorded too.
2. In **Runs**, mark the working run **R** (reference) and the failing run **F**.
3. Open **Compare**.

It matches requests and scripts by path, ignoring host and folder, and shows the first request or script the working copy had that the failing run didn't, requests that failed only in the failing run, errors only there, and functions and branches that ran in one and not the other.

“The first thing that differed” is often the whole answer.

You don't need a real LMS to practise. Any two runs of the same item can be compared, one marked **R** and one **F**.

Try it: compare two local runs

1. Open `06-stuck-connecting.zip`, pin `startOK` to `true` (chapter 8), turn on **Trace the code**, and run for 10 seconds. This is your working copy: mark it **R**.
2. Remove the pin and run again, still traced. Mark this one **F**, and open **Compare**.
3. What does the comparison list as the first difference? Which functions ran only in **R**? Does it point at the same line you found in chapter 8?
4. Turn **Trace the code** off again.

Choosing the tool

You know	Use
Nothing failed, it just stopped	The **waiting for** finding, then **Drill-down**
Which line decided, but not why	**Drill-down**, or trace and read **Execution** from the bottom
It asks a server for something	**Server needs**
It works somewhere else	**Record a working copy** and **Compare**
It fails only in WebKit	Reproduce in Chromium with the same storage and device, then trace that

Check yourself

1. What does “waiting for: a reply to *player-ready*” tell you, and what question does it make you ask?
2. Why read the Execution view from the bottom up?
3. A drill-down chain ends with a warning, not an error. What does that tell you?
4. Server needs says the course read nothing from a response. Is that useless?
5. A static finding said a script would fail, but the run shows it didn't. What do you do with the finding?

Next: Delivering the answer.

13. Delivering the answer

What you'll learn: how to turn a diagnosis into something useful to someone else: a corrected package, a report that leaves the course content out, a saved run, a comparison of two exports, and a check of a whole folder.

A diagnosis that stays in your head helps nobody. The author needs to know what to change in their tool, the LMS administrator what to change in their settings, the vendor what their product does wrong. Each needs evidence they can check, without your having to send them a course they may not be allowed to see.

A corrected package

Some problems have a mechanical fix: one Package Lab can make without guessing what the author meant. **Export corrected package...** (in **Actions**, or **File › Export Corrected Package...**, ⌘E) shows a checklist of the fixes that apply, all ticked. Untick any you don't want and choose **Export...**.

Fix, as the checklist names it	What it does
Rename the manifest to imsmanifest.xml	`Imsmanifest.xml` becomes `imsmanifest.xml`
Move the package's files to the top of the ZIP	For a ZIP made from the folder above the package
Remove manifest file entries for files that aren't in the package	Takes `<file>` lines for absent files out of the manifest
Make the manifest's SCORM version match the API the scripts use	Sets the manifest's version and namespaces
Make references match the capitals of the files	Changes references to match the file, or renames the file when every reference spells it the same way
Load scripts, styles and frames over https	Changes `http://` scripts, styles and frames to `https://`
Remove Xyleme's server-side player setting	Removes the `PLAYER_TYPE = 'Server-side'` line that makes a Xyleme export wait for Xyleme's server

The export also includes **every file you changed in Files**. So a fix you proved with an override in chapter 8 can be exported as a package.

Package Lab never touches the original. The new ZIP has a `PACKAGE-LAB-CHANGES.txt` listing every change, and anything it decided not to do. The status line says how many findings the changes cleared: it checks the new package again, from its files.

A corrected package is a test, not a release. Run it, then try it in the real LMS. And send the author the list of changes, so the next export from their tool doesn't bring the problem back.

Try it

1. Open `03-not-tracked.zip` and export a corrected package. Which fix is offered? Open the result and run it. Does it track now?
2. Open `02-blank-page.zip` and export. Read `PACKAGE-LAB-CHANGES.txt`. The capitals fix renamed a *file* rather than the reference. Why? Is the package now fixed? What's left, and why can't Package Lab fix it?
3. Open `04-loading-forever.zip` and try to export. What does Package Lab say? Fix it yourself: correct `data/config.json` in **Files**, save it as an override, run to prove it, and export again.

A report without the course content

Courses are often confidential, licensed, or full of learners' answers. A redacted report shows what went wrong without any of that.

1. Select the run that shows the problem.
2. On **Findings**, tick **Take the course content out**.
3. Choose **Save report...**.

What goes: the text the course shows, console messages, stored values (answers, names, bookmarks), suspend data, request and response bodies, typed text and macros, changed file contents, the scripts' text, xAPI statement contents, query values, local file paths and the item's title.

What stays: the verdict, the findings (worked out again from what's left), milestones, the timeline, API calls with their element names, numbers and times, function and property names, the package's name and file paths, and the first part of each error message. File and folder names can carry a client's or a course's name: the command line's `--redact-names` hashes them.

A banner at the top of the report says what was taken out. **Read the report before you send it**: a browser's error message can still carry words the course wrote. *(Command line:)* `report --redact --redact-names --key-out` also replaces the file names with hashes, and gives you the key to match them back.

Saved runs

Every run is saved as JSON in the data folder's `runs/` folder. **File › Open Saved Run...** (or **Open saved...** under **Runs**) brings one back, to read its views or to compare it with another run. Opening it doesn't change your settings or package.

A saved run also records the launch options it used: settings, changed files, pins, answers and macro. The command line can repeat it exactly *(command line, chapter 14; the app has no button for this)*:

```
package-lab run course.zip --options-from run.json
```

The run doesn't contain the package itself. To hand a diagnosis to a colleague who has Package Lab, send them the saved run *and* the same ZIP. They can read your run in the app, and repeat it, or change one thing, with `--options-from` (chapter 14). Saved runs hold the course's scripts and stored data, so treat them like the course.

Comparing two packages

Two exports of the same course, one that works and one that doesn't? **Compare packages...** (in **Actions**) asks for a second ZIP and compares it with the open one, from the files, without running either. The **Package diff** view shows:

- **Files**: added, removed and changed, each text file with a line diff;
- **Manifest**: what differs, field by field;
- **Loading path**: files only one of them loads, and files whose state changed (*in package* in one, *missing* in the other);
- **New findings and findings gone.** Start with the new ones.

Try it

Open `01-works.zip`, choose **Compare packages...**, and pick `02-blank-page.zip`. Which files changed? What do the new findings say? Could you have diagnosed `02-blank-page.zip` from this view alone?

Checking a folder

A course often arrives as a folder of exports, one per language or per LMS. **Check a folder...** (in **Actions**) checks every ZIP in a folder, from the files, and shows one row per package in **Folder**

check: its verdict, SCORM version, tool, counts of errors and warnings, and the worst finding. Click a row to open that package.

Try it

Check your practice folder. How many packages does it say are **OK**? How many of those have you found faults in?

This is the most important lesson in this chapter. A folder check is fast and catches the problems the files can show: wrong names, missing files, versions, capitals, server paths. It **cannot** see a course that waits for a message, keeps state in blocked storage, ignores its bookmark or fails only in Safari. Most of your practice packages pass a folder check, and of those only the three with nothing wrong in the course (01, 14 and 15) work. Use a folder check to triage, then run the ones that matter.

Writing it up

Whatever you send, the reader needs the same four things, and Package Lab's findings are written in this order for that reason:

1. **What happens**: the symptom, in the learner's terms.
2. **Where it stops**: the milestone, and the finding.
3. **The evidence**: the file and line, the request, the API call. Attach the redacted report.
4. **What to change, and the proof that it works**: "With `startOK` pinned to `true`, the course tracks normally (run attached)."

The last part is what makes the report land. "I think it's the manifest" starts an argument. "Changing the manifest's version to 1.2 makes it track, here's the run" ends one.

Check yourself

1. Which problems can a corrected package fix, and which can't it?
2. Why must you still test a corrected package in the real LMS?
3. Name three things a redacted report removes, and three it keeps.
4. A folder check says a package is OK. What does that tell you, and what doesn't it?
5. Write a four-part report for any one practice package you've diagnosed.

Next: Working at scale.

14. Working at scale

What you'll learn: how to do everything you've done in the window from the command line, so you can check many packages, repeat a diagnosis exactly, script it and keep the results. And how to use Package Lab from another device.

The desktop app is the best place to explore one course. When you have twenty courses, or the same check to run every time a vendor sends an update, or a colleague who needs to repeat exactly what you did, the command line is better.

Before you start: the source code

Everything else in this guide works in the downloaded app. The command line and the web interface in this chapter, and the WebKit runs in chapter 11, don't: they need Package Lab's **source code**, the developer copy of Package Lab. It is a folder of files rather than an app, and it runs with Node.js. If you don't have the source, read this chapter for interest. The app does most of it: **Check a folder…** does the same as `batch`, and the analyses have buttons in the window. What only the command line can do: repeat a saved run exactly (`--options-from`), run under a profile of your own LMS (`--profile-file`), print a macro (`macro show`), hash file names in a report (`--redact-names`), and give results to scripts (`--json`, `--strict`). Each is marked *(command line)* where the guide mentions it.

If you do have the source, set it up once. You need Node.js 20 or later: open **Terminal** (Applications › Utilities), type `node --version` and press Return. If it says "command not found", install Node from its website first. Then type `cd` and a space, drag the source folder from Finder onto the Terminal window, and press Return. Now run these, one at a time:

```
npm install
```

```
npm link
```

The first installs what Package Lab needs, which takes a minute or two. The second makes the `package-lab` command work in any folder. To check everything is in place:

```
package-lab doctor
```

It ends with a **To do** list of anything missing. On Windows, `docs/windows.md` in the source explains the same steps.

Add WebKit

Only for the WebKit runs in chapter 11. In the source folder:

```
make webkit
```

That downloads Safari's engine. WebKit runs then work when you open Package Lab from the source, with `package-lab open`, not from the app in Applications.

The command line in five commands

Every command starts with `package-lab`. Add `--help` to any of them for its options.

```
package-lab analyze course.zip
```

The static findings, from the files only. Fast, and needs no browser.

```
package-lab run course.zip --wait 20 --out run.json
```

Runs the first SCO for 20 seconds and prints the milestones and findings. `--out` saves the run, which you can open in the app with **File › Open Saved Run...**.

```
package-lab report course.zip --run --open
```

Runs it and opens an HTML report in your browser: findings, milestones, timeline, and code excerpts for each `file:line`. Add `--redact` to take the course content out.

```
package-lab fix course.zip --out fixed.zip
```

Writes a corrected package, as **Export corrected package...** does.

```
package-lab batch ~/Downloads/vendor-exports --csv results.csv --out results.html
```

Checks every package in a folder: one row each, as a table, a CSV and an HTML report. Add `--run` to run each one too, and get the furthest milestone of each.

Try it

Give it the practice kit's folder. On a Mac that is:

```
package-lab batch ~/Library/Application\ Support/Package\ Lab/practice --run --wait 8
```

On Windows, use `%APPDATA%\Package Lab\practice`. **Show folder**, under the guide's contents, opens it if you're not sure.

It takes a couple of minutes. Compare the **Furthest milestone** column with the **Verdict** column, and with what you've learnt about each package. Which packages does `--run` catch that the plain batch in chapter 13 didn't?

Every setting is a flag

Every setting in the inspector has a flag, so any run you did in the app can be repeated here:

In the app	On the command line
Item	`--item 2`
API	`--api both`
Profile	`--profile moodle`
Frame depth, In a new window	`--depth 2`, `--window`
Stop after	`--wait 20`
Trace the code	`--trace`
Storage, Browser, Device size	`--storage blocked`, `--browser webkit`, `--viewport ipad`
Authoring tool	`--tool storyline`
Pins, Answers	`--pin previewMode=false`, `--answer '/api/config=200:{}'`
Changed files	`--override js/app.js=fixed-app.js`
Macro	`--macro macro.json`
Resume last run's data	`--resume-from run.json`
Record caught errors (off)	`--no-caught`
Keep typed text in runs	`--record-input`
Watch the course window	`--show`

There's no flag for **Stop after: I stop it**: a command-line run always lasts `--wait` seconds. (`--live` lets another program watch and drive a run.)

You don't need to remember them. **Copy as CLI command**, at the bottom of **Settings › Advanced**, copies the `package-lab run` line for the current settings. The command line reference has the full list.

And the tools of the Deep dive and Tools groups are commands:

In the app	Command
Experiments	`package-lab experiment course.zip --preset lms`
Server needs	`package-lab discover course.zip`
Drill-down	`package-lab drill course.zip`
Test saving	`package-lab cycle course.zip --macro macro.json`
Compare packages	`package-lab diff a.zip b.zip` (`--run` runs both)
Check a folder	`package-lab batch folder` (`--recursive` for subfolders, `--jobs n` to run several at once)

Some of them do more on the command line than in the window:

- `experiment --variants file.json` runs *your own* variants: a list of `{ "name", "patch" }`, each changing one thing. The recipe for one LMS and not another shows it finding which field of a profile matters.
- `discover --request /api/player/config --rounds 3` works on one request, for at most three rounds.
- `items course.zip` lists a package's launchable items, numbered for `--item`.

Options files: a diagnosis you can repeat

When a run needs several pins, answers and changed files, put them in an **options file**, a JSON file:

```
{
  "expose": "both",
  "pins": [{ "path": "previewMode", "value": false }],
  "responses": [{ "match": "/api/player/config", "status": 200, "body": "{}" }]
}
```

```
package-lab run course.zip --options fix-test.json --wait 20
```

The names in the file aren't always the flags' names (the **API** setting is `expose`, for example). The easy way to get them right is to set the run up in the app, then look at **The options as JSON** under **Settings › Advanced**, or at `launch.options` in a saved run.

And because every saved run keeps its options, `--options-from run.json` repeats a saved run exactly. That is how you hand a diagnosis to a colleague, or re-check it after the vendor's next export.

Profiles of your own

Once you know how your real LMS hosts courses (ask its administrator), write it down as a profile file: any fields, over a preset named in `base`.

```
{ "base": "moodle", "contentOrigin": "other", "frames": { "depth": 2 }, "limits": { "suspend12": 4096 } }
```

```
package-lab run course.zip --profile-file our-lms.json --wait 20
```

Now “does it work in our LMS?” is a command you can run on every course before it's uploaded. The fields are in the command line reference.

JSON output and scripting

Every command takes `--json` and prints its result as JSON, for scripts. Exit statuses tell a script what happened:

Status	Means
0	The command worked (the findings may still include errors)
1	With `--strict`, the findings include at least one error
2	The command failed: bad arguments, an unreadable file, a run that didn't finish

So `--strict` gates on *errors*. A run that only stops short of a milestone, with warnings, still exits 0. To gate on how far a run got, read `furthest` from the JSON (`launched`, `scripts`, `shown`, `looked`, `found`, `initialised`, `read`, `wrote`, `committed`, `finished`):

```
package-lab run course.zip --profile-file our-lms.json --wait 20 --json | jq -e '.furthest == "committed" or .furthest == "finished"'
```

`jq -e` exits 1 when the answer is false, so a vendor's package that doesn't reach **Committed** (or go past it to **Finished**) under your LMS's profile can stop a pipeline before it's uploaded. `jq` is a separate download from jqlang.org, on the Mac or Windows.

The web interface: Package Lab from another device

```
package-lab serve
```

This prints an address with a sign-in token, such as `http://127.0.0.1:4880/?token=…`. Open it in a browser: it's nearly the whole app, plus a **Live** view of the course window that you can click and type into. Courses still run on the computer running `serve`. The differences that matter:

- **Open package…** uploads a ZIP from the device you're on, and it replaces the package open on the Mac *for everyone* using it.
- There's no **Check a folder…** (use `package-lab batch` on that computer), no **Record** button and no menu bar. Record macros with **Record clicks** in the Live view.
- Settings, pins, answers and the macro are kept in that browser's page: reloading clears pins, answers and the macro.

The web interface guide lists every difference.

To reach it from an iPad or another computer, see Reaching it from another device. Read the security notes first: anyone with the address and token can use it, and the token is in the address, so it can end up in a browser's history. Don't paste it into chat or a ticket.

Check yourself

1. How do you repeat, on the command line, a run you set up in the app?
2. A colleague can't reproduce your result. What do you send them?
3. Write the command that runs a course under your own LMS profile and fails if it doesn't reach **Committed**.
4. Why might you run `batch --run` rather than plain `batch` on a vendor's delivery?

Next: Thinking like an analyst.

15. Thinking like an analyst

What you'll learn: the method that ties every chapter together, the habits that keep you honest, the traps that catch experienced people, how to read authoring-tool and xAPI findings, and a final challenge.

You now know every tool. What makes an expert isn't knowing more buttons. It's knowing which question to ask next, and not believing an answer until it's proved.

The method

Every diagnosis, however hard, follows the same loop.

1. **Get the symptom in the learner's words.** Where (which LMS, browser, device), when (at launch, on page 5, on return), and what they see. Most wasted hours start with a vague symptom.
2. **Read the files.** Static findings. Note them as predictions.
3. **Reproduce it.** Run with the defaults. If it works, change the environment towards the learner's: profile, browser, device, storage. **You can't fix what you can't reproduce.** If you can't reproduce it in Package Lab, say so, and record a working copy in the real LMS (chapter 12).
4. **Locate it.** The first missed milestone. The first problem in loading order. The first error in time order.
5. **Explain it.** Follow the evidence to a cause: findings, then views, then waiting-for, drill-down and tracing. Stop when you can say "it stops at *X* because *Y*, shown by *Z*".
6. **Prove it.** Change one thing (a setting, a file, a pin, an answer) and show the course gets further. If it doesn't, your explanation was wrong: go back to step 5.
7. **Hand it on.** What happens, where it stops, the evidence, and the change that fixes it with its proof (chapter 13).

A map from symptom to first move

Symptom	First move	Then
Blank page	Run; read the blank-page finding's evidence in order	Boot path's first *never asked for*; Errors' first error
Stuck on loading	Run; read the loading-screen finding	Errors; **waiting for**; Timeline around the stop

Symptom	First move	Then
Not tracked	Run; Lookups	**API: Both**; **frames** experiment; Package's mode switches
Doesn't resume	Record a macro; **Test saving**	The first resume finding; API of run 2
Fails in Safari or on an iPad	Run with **WebKit** and **iPad**; **Storage: Blocked**	Errors; reproduce in Chromium and trace
Fails in one LMS	**lms** experiment	One profile field at a time
Worked in the authoring tool	Run; Network for server-root requests	**Server needs**; answers; ask for an LMS export
Nothing fails, nothing happens	**waiting for** finding	**Drill-down**; trace; **Execution** from the bottom

The recipes are the long form of six of these rows.

Habits

- **Glance at the summary chip before every run.** A setting left from the last test is the commonest cause of a confusing result. So is a macro you forgot to clear, and a pin or an answer left in **Settings › Advanced**. Most settings survive quitting the app, so check after a restart too. (Pins, answers and the macro don't.)
- **Name what changed.** One line per run in a log: the change, and how far it got.
- **Start each theory from a clean baseline.** A run with the defaults, nothing changed, to compare against.
- **Give the course a fair chance.** A 10-second run with nobody clicking can't call `LMSFinish`. Use **I stop it** and leave the course properly before blaming it for not finishing.
- **Read the findings, not only the banner.** The milestone can stay the same while the cause changes (chapter 8).
- **Keep the evidence.** Save runs, keep `--out` folders, keep redacted reports. You'll be asked again.

Traps

Static findings are predictions. `05-error-in-the-console.zip` carries a static Problem, “Scripts run before what they need”. The error does happen, and it doesn't matter: the course starts properly a moment later. Severity says how bad a problem would be, not that it is. And the reverse: most of the practice packages have no static findings at all, and every one of them except the three sound ones is broken.

A finding's Try: is the commonest fix, not this course's fix. In `13-keeps-looking.zip` the finding suggests trying both APIs. That's right for most courses that show it, and wrong for that one.

A folder check can't see run-time faults. Most of the practice packages pass one, broken or not.

Package Lab's LMS isn't your LMS. Profiles are approximations from public documents, and administrators change settings. A run is a strong hint, never a verdict. Confirm every fix where the learners are.

Absence of evidence. A WebKit run has no caught errors and no coverage; a trace leaves out `eval` and HTML `onclick` handlers; drill-down can't watch a function's local variables; server needs can't see inside a worker. When the evidence is silent, check whether the tool could have seen it.

Changing two things. If it works after two changes, you don't know which one mattered, and you may tell an author to change something that didn't need changing.

Authoring tools

Package Lab knows eleven authoring tools and wrappers: Xyleme, Articulate Storyline and Rise, Adobe Captivate, iSpring, Lectora, Adapt, dominKnow, Evolve, the SCORM Driver and the pipwerks wrapper. When it recognises one, it adds findings that start with the tool's name (“Adapt: …”), and each says how sure it is:

- **Matches a known pattern**: from forum reports and the like. Information, worth checking.
- **Confirmed in the package's files**: Package Lab read the setting itself.
- **Confirmed in this run**: the recording shows it.

A package that doesn't name its tool is treated like any other, unless you set **Authoring tool** in **Settings › Advanced** (or `--tool` on the command line). The authoring tool notes say what Package Lab knows about each, and where it comes from. Read the one for the tools your organisation uses: knowing how a tool's exports usually fail is half the diagnosis.

cmi5 and xAPI courses

A cmi5 or Tin Can package opens like any other. What changes:

- **There's no SCORM API.** The test LMS gives the course a learning record store instead, and puts its address in the launch URL, as an LMS does. **API**, **Resume last run's data** and **Test saving** don't apply.

- **The milestones are different.** For cmi5: launch page loaded, scripts loaded, content shown, fetched the auth token, initialized, completed/passed/failed, terminated. For Tin Can: the first three, then sent a statement and sent a completion statement.
- **The xAPI view** replaces the API view: every statement in order, who sent it, whether the LRS stored or refused it, and its validation problems, each with its place in the statement (`result.score.scaled`); the state and profile documents; and every request to the LRS.

The LRS checks every statement against xAPI 1.0.3, and the cmi5 rules. A statement that breaks a MUST is refused with status 400, as the specification requires. Real LRSs vary: some are stricter, some store what they should refuse, so a statement your LRS accepted isn't proof it's valid. Most cmi5 faults are about order and context: “completed” before “initialized”, a missing session id, the auth token fetched twice. The findings list them all. Profiles, frame depth, device sizes, storage and WebKit all still apply.

cmi5 in one paragraph. A cmi5 course is made of **AUs** (assignable units, cmi5's word for a launchable item). The LMS launches an AU with its LRS address, an actor, a **registration** (the learner's enrolment) and a one-use URL to fetch an **auth token**. The LMS sends a “launched” statement itself. The AU then fetches the token (once only), sends “initialized”, then “completed”, “passed” or “failed” as they happen, and “terminated” last. Every statement in that **session** carries the same session id in its `context`. The course's `moveOn` rule, in the package's `cmi5.xml`, says which of completed and passed the LMS needs before it counts the AU as done.

Try it: `16-cmi5-never-passes.zip`

“The LMS shows the lesson completed, but never passed.”

1. Open it. What kind of package does the window say it is? What does **Package** list as the items?
2. Run it for 10 seconds. Read the banner. Did it reach every milestone?
3. Now read the findings. How many errors are there, when every milestone has a tick?
4. Open **xAPI**. Which statement was rejected, and what is the problem with it, at which place in the statement?
5. What else is wrong with the order of the statements, and what would a cmi5 LMS do about it?

Try it: `17-tincan-answer-missing.zip`

“The completion arrives, but the quiz answer never shows in our reports.”

1. Run it for 10 seconds. Which milestones did it reach?
2. In **xAPI**, find the statement that was refused. What value broke which rule?
3. What happens to a refused statement in a real LRS, and so in the learner's record?

Check your answers in the answers.

The lesson of both: **milestones measure how far a run got, not whether everything it sent was accepted.** For xAPI, always read the xAPI view.

The final challenge

You've diagnosed every practice package one chapter at a time, knowing which tools each chapter was about. Real courses don't come with a chapter. Three ways to finish:

1. **Cold diagnosis.** Ask someone to rename the nineteen practice packages to `a.zip` to `s.zip` without telling you which is which. Diagnose them all with the method above, then check against the answers.
2. **Make a fault.** Open `01-works.zip`, change one file in **Files** to break it in a way this guide hasn't shown (a typo in an element name in `lesson.js`, `cmi.core.lesson_staus`, say; or `LMSCommit` removed), export the package, and give it to a colleague to diagnose. Then do the same with theirs. Before you hand it over, check that your fault is findable: run it yourself, and note which milestone, finding or Test saving result shows it. Mark your colleague's diagnosis against the four parts of a report: the symptom, where it stops, the evidence, and a change that's proved to fix it.
3. **A real course.** Take a course that has given your organisation trouble. Follow the method from step 1. Write the four-part report. If you can't reproduce it, write that down too, with what you tried: that's evidence as well.

Beyond this guide

Some things an expert meets that this guide only touches:

- **Interactions and scores.** The sample course records each quiz answer as `cmi.interactions.n.*`, and compares its score with the manifest's mastery score (`cmi.student_data.mastery_score`). It also reads `cmi.launch_data`, from the manifest's `adlcp:datafromlms`. An LMS's reports of answers come from these. Read them in the **API** view.
- **SCORM 2004's two statuses.** 2004 splits 1.2's `lesson_status` into `cmi.completion_status` (completed or not) and `cmi.success_status` (passed or failed), and adds `cmi.score.scaled` (-1 to 1). A course that sets one and not the other shows up oddly in LMS reports.
- **Sequencing.** SCORM 2004 can carry rules for the order of SCOs and what unlocks what. Package Lab launches one item at a time and doesn't play sequencing rules: test those in the LMS.

- **Several SCOs, and how the LMS reports them.** How an LMS rolls several SCOs' statuses up into one course result, and how it treats `cmi.core.lesson_mode` and `credit`, are the LMS's business. Package Lab shows what each SCO sends; what the LMS's reports make of it, test in the LMS.
- **Authoring tools.** The authoring tool notes, one page per tool.
- **The Windows build.** `docs/windows.md` in the source code covers building and running it.

Where to go from here

- The recipes: step by step for the commonest problems.
- Findings: every finding, what it means, what to do.
- The Help: every view, button and setting.
- The command line reference: every command and flag.

Check yourself

1. State the seven steps of the method from memory.
2. You can't reproduce a learner's problem in Package Lab. What do you do?
3. Name three things you should check before every run.
4. What does “Confirmed in this run” on an authoring-tool finding mean, compared with “Matches a known pattern”?
5. A cmi5 course's LRS refuses a statement with status 400. Where do you look, and what will it tell you?

Appendix A: Exercise answers

What is wrong with each practice package, how Package Lab shows it, and what fixes it. Try each exercise before you read its answer: the point is the method, not the fault.

The outputs quoted here come from 10-second runs with the default settings. Times and counts in yours may differ a little.

01-works

Nothing. It is the sample course as it was written. A 10-second run nobody uses stops at **Committed**, with “The content never called LMSFinish” and “The content never set exit”, because nobody left the course. Used and left with **Save and exit**, it runs through. With a macro that moves two pages, **Test saving** ends with “Saving and resuming worked”.

02-blank-page

The fault: the launch page has no content of its own; a script builds it. The page asks for `js/app.js`, but the file in the package is `js/App.js`. It also asks for `vendor/missing.js`, which isn't there at all.

How it shows: static Problems “1 file the launch loads isn't in the package” and “Paths differ from the files only in capitals”, and the note “The launch page has no content of its own”. The run stops after **Launch page loaded**, with “The content shows a blank page”, whose evidence starts with `Not found: js/app.js (404)`.

Which matters: the capitals first. `js/App.js` is the script that builds the page, so without it nothing shows. The missing `vendor/missing.js` is a real fault too, but nothing on the page depends on it.

The second fault: fix the capitals and run again, and the page is *still* blank. Now the first cause in the blank-page finding is a new one: `TypeError: Cannot set properties of null (setting 'innerHTML')`, at `js/App.js:1` if you corrected the reference in `lesson.html` (chapter 8's way), or `js/app.js:1` if **Export corrected package...** renamed the file. The script runs in the page's `<head>`, before the `<body>` exists, so `document.body` is `null`. The first fault was hiding the second. This is normal: fix the first cause, run again, and read the first cause again, until the run gets where it should.

The fix: make the reference match the file name. **Export corrected package...** offers this, and because the only reference spells it `app.js`, it renames the file. Then run the script after the body

exists: add `defer` to its `<script>` tag, or move the tag to the end of the body. And republish with `vendor/missing.js`, or remove the reference.

With the first two as overrides, the page shows “Built by App.js”, with no error. **Content shown** still has a cross, because it needs at least 20 characters of text and the sample page has 15: milestones are rules of thumb, too. This sample page has no SCORM code of its own, so even fixed it never looks for the LMS. That's a limit of the exercise, not of your diagnosis.

03-not-tracked

The fault: the manifest declares SCORM 2004, but the scripts use the SCORM 1.2 API.

How it shows: static Problem “The manifest says SCORM 2004, but the scripts use the SCORM 1.2 API”. In the run, the course shows its pages and its first line says “No LMS found: this page works, but nothing is tracked.” It stops at **Looked for the API** with “The content looked for API and found nothing: This run offered API_1484_11: the package and its manifest disagree on the SCORM version.” **Lookups** shows it looking for `API`.

The proof: with **API: Both**, or the **api** experiment, it reaches **Committed**: “api both gets further than the default”, and so does “api 1.2”. “api 2004” and “api none” make no difference.

The fix: either republish for SCORM 2004, or correct the manifest to 1.2. **Export corrected package...** makes the second (“Match the SCORM version”), and clears the finding.

04-loading-forever

The fault: `data/config.json` has a trailing comma (`{ "start": true, }`), which isn't valid JSON. `js/boot.js` reads it and parses it before it loads the next part of the course; the parse throws, so the next part never loads and the loading screen stays.

How it shows: static Problem “data/config.json isn't valid JSON”, read from `js/boot.js:2`. The run stops after **Scripts loaded** with “The content is stuck on a loading screen (div#loading.spinner)”, whose first evidence is the `SyntaxError` at `js/boot.js:2`. “Code that loads the next part of the content never ran” names `js/next.js`.

The fix: remove the comma. Package Lab has no automatic fix for this (it can't know what the author meant), so **Export corrected package...** says there's nothing to change. Correct it in **Files**, save it as an override, run to prove it, then export.

05-error-in-the-console

The fault: an inline script on the launch page calls `scorm.start()` before `scorm.js` has run (it's loaded with `defer`, so it runs later). That throws `ReferenceError: scorm is not defined`.

How it shows: static Problem "Scripts on lesson.html run before what they need". The run shows the uncaught error at `lesson.html:7`, and then reaches **Committed**, exactly as `01-works.zip` does.

Does it matter? Hardly. The course's own `lesson.js` starts the session properly a moment later, so it tracks. The answer to "does it matter?" is: it's a real error, it should be removed because it's noise that hides real problems, but it isn't why anything fails. The static finding's severity said how bad it *could* be; the run shows it isn't.

06-stuck-connecting

The fault: the last line of `lesson.js` is `if (window.startOK) start()`, and nothing sets `startOK`. So `start()`, which finds the LMS, never runs, and the page keeps its first message, "Connecting to the LMS…".

How it shows: no static findings. The run stops after **Content shown** with "The content never looked for the SCORM API", and no error. Coverage in `lesson.js` shows the `if` ran and `start()` didn't. Drill-down says: "In lesson.js:68, `if (window.startOK)` was false because `window.startOK` was undefined", then "`startOK` was never assigned (watched), and nothing in the package sets it. It may be something the LMS or the page around the course is expected to provide."

The proof: pin `startOK` to `true`, or change the line to `start()` as an override: either way it reaches **Committed**.

The fix: ask the author what `startOK` is meant to be set by. Courses that expect their player or a wrapper page to set a flag are common; the fix is an export without that dependency.

07-works-on-the-platform

The fault: the course asks `/api/player/config` (a path at the root of the server, which only its authoring platform has). When that fails, it creates its own `window.API`, a preview stand-in that answers `true` to everything, and talks to that instead of the LMS.

How it shows: static Problem "The launch depends on paths at the root of the server", and warnings "Scripts create their own SCORM API object" and the mode switch `previewMode`. The run

looks good at a glance (it reaches **Initialised**), but its first finding is “The content set its own window.API or API_1484_11… 1 API call went to it”. Network shows `GET /api/player/config` failed.

The proof: answer `/api/player/config` with `{}`. The milestone stays the same, but “The content set its own window.API” goes, and the `LMSInitialize` goes to Package Lab's LMS. (It reads and writes nothing after that, because this sample's lesson starts before the answer arrives; in a real course, this is where you'd use **Server needs**.) **Server needs** reports that the course read nothing from the response: any successful reply would do.

The fix: an export for an LMS or standalone, without the platform dependency. Ask the author for it, with the redacted report as evidence.

08-forgets-its-place

The fault: `scorm.js` never sets `cmi.core.exit`, so when the learner leaves early, the LMS isn't told they're suspending. Package Lab's test LMS, like many real ones, then starts a new attempt.

How it shows: a timed run is identical to one of `01-works.zip`, notes and all: nobody leaves, so neither sets exit. Only **Test saving** tells them apart. It shows **Exit set** empty in run 1 and **Entry offered** empty in run 2, with “The LMS started a new attempt instead of resuming (entry was “”)”.

The fix: set `cmi.core.exit` to `suspend` whenever the learner leaves before finishing.

09-starts-over

The fault: on resume, `lesson.js` reads `cmi.core.lesson_location`, then ignores it and opens page 1.

How it shows: nothing in a single run. **Test saving** with a macro that moves on a page or two shows **Location saved (first, last)** of `1, 3` in run 1 and `1, 1` in run 2, with “The content read its saved place but started from the beginning: left at “3”, first saved after resuming: “1””. Without a macro, run 1 never leaves page 1, so the test passes: the fault only shows when the learner has somewhere to come back to.

The fix: use the location that was read to open the right page.

10-fails-in-safari

The fault: the first line of `lesson.js` reads `navigator.userAgentData.brands`, which only Chromium-based browsers have. In Safari's engine `navigator.userAgentData` is undefined, so the line throws and the rest of the script, which starts the course, never runs.

How it shows: fine in Chromium. In WebKit, it stops after **Content shown** with “The content never looked for the SCORM API” and the uncaught `TypeError: undefined is not an object (evaluating 'navigator.userAgentData.brands')` at `lesson.js:1`.

Each run: in Chromium it reaches **Finished**. In WebKit at the iPad size it stops after **Content shown**, with the error above. WebKit alone at the normal size stops in the same place: the fault is the engine, not the screen, so **Device size** makes no difference here. The **browsers** experiment agrees: Chromium reaches **Finished**, WebKit stops after **Content shown**.

The fix: check that `navigator.userAgentData` exists before using it, or don't use it.

11-fails-in-some-lmss

The fault: a script reads and writes `localStorage` before the lesson starts, and the lesson only starts if that worked. Where storage is blocked (Safari, and browsers that block third-party cookies, for content in another site's frame), reading `localStorage` throws, so the lesson never starts.

How it shows: fine with the defaults and most profiles. The **lms** experiment shows **scorm-cloud** getting less far (to **Content shown**), with “With storage blocked, the content gets less far”: the scorm-cloud profile blocks third-party cookies. With **Storage: Blocked** alone, the same, and “The content needs storage the browser can block” names `localStorage` at `boot.js:2`.

The fix: keep the learner's state in the LMS (`cmi.suspend_data`), and wrap any browser storage in `try/catch` so the course carries on without it.

12-waits-forever

The fault: `boot.js` tells its parent window it's ready (`{ type: 'player-ready' }`) and waits for a reply (`{ type: 'launch' }`) before it starts the LMS session. That's a handshake with a particular player; no LMS sends it.

How it shows: no static findings, no uncaught error and no failed request. `scorm.start()` is on `boot.js:2`, inside the message listener, so it would run only when the message came. It stops after **Content shown**, with “The content was waiting for: a reply to “player-ready”; a message to arrive (1 listener never ran)”, whose evidence is “boot.js:1 sent “player-ready” to the LMS window (origin *): nothing there was listening” and “boot.js:2 listens for a message that never came”.

The proof: a pin can't send a message. Change `boot.js` in **Files** so it starts the session once the page has loaded, `addEventListener('load', function () { scorm.start() })`, without waiting for the message. Save it as an override and run: it finds the API and initialises. (Calling

`scorm.start()` at once won't do: `boot.js` runs before `scorm.js`, so `scorm` doesn't exist yet: Package Lab's static check would then report “Scripts run before what they need”.)

The fix: an export that doesn't expect the player's handshake, or an LMS-side wrapper that answers it.

13-keeps-looking

The fault: `boot.js` checks `window.API` every 250 ms, on the course's *own* window, and only starts the session when it's there. An LMS puts the API on one of its own windows, a parent frame or the opener, never on the course's window, so it never appears.

How it shows: no static findings. The run stops at **Looked for the API** with “The content looked for API and found nothing: It looked in frames where the LMS hadn't put the API”, and “The content was waiting for: a timer at boot.js:1 (every 250 ms)”. **Lookups** shows every lookup on `lesson.html`. Drill-down: “In boot.js:2, `if (window.API)` was false because `window.API` was undefined”, then “`API` is the LMS's SCORM 1.2 API: it lives on an LMS window (a parent frame or the opener), not on the content's own window.”

The trap: the finding's **Try:** suggests **API: Both**. That doesn't help: the API isn't missing, it's somewhere the course doesn't look. Frame depth and a new window don't help either.

The proof: change `window.API` to `parent.API` on line 2 of `boot.js`, as an override. It finds the API and initialises.

The fix: search for the API the standard way: the window, then each parent up to the top, then the opener.

14-next-button-late

Not a fault in the course: the exercise is the macro. **Next** is on the page from the start but hidden for 3 seconds, so the macro's clicks at 0.5 s and 0.9 s find no visible element, fall back to their position, and hit nothing. A 10-second run leaves `cmi.core.lesson_location` at `1`.

The fix: a wait for the text `Next` before the first click. With it, the clicks land once the button shows, and the location goes `1`, `2`, `3`. The same is true of real courses with intros, animations or slow requests: wait for something, not for a time.

15-button-moves

Not a fault in the course: Next is pinned 20 pixels from the top right corner, so its place across the window changes with the width.

- As recorded, and at **iPhone** size, the clicks find `#next` and click it wherever it is: the location reaches `3`. At iPhone size, the note “The macro was recorded at 1100×800 and played at 390×844: clicks may miss” warns you, but they don't miss.
- With the selector changed to `#nope`, the clicks can't find their element and use their recorded position, and the run notes “2 clicks didn't find their element and used their recorded position”. At iPhone size that position happens to land on the button anyway, which is luck, not design.
- At 2400 × 800 the recorded position (93.6% across, 2,246 pixels) is left of the button (2,280 to 2,380 pixels), so the position-only clicks miss, and the location stays at `1`.

The lesson: a click tied to its element survives layout changes; a position alone works only at the size it was recorded.

16-cmi5-never-passes

The fault: the AU sends its statements in the wrong order (“completed” before “initialized”), and its “passed” statement uses the bare word `passed` as its verb id instead of the IRI `http://adlnet.gov/expapi/verbs/passed`.

How it shows: no static findings. The run ticks every cmi5 milestone, and the banner says “Reached every step”. But there are three errors: “The LRS rejected 1 statement” (`#4 verb.id: verb.id isn't an absolute IRI`), “1 statement has a verb id that isn't an IRI”, and “cmi5: The AU sent “completed” before “initialized””. The **xAPI** view shows statement 4 as rejected.

Why the LMS never shows passed: the LRS refused the “passed” statement with status 400, so it was never recorded. A cmi5 LMS may also ignore or reject the whole session for its order. (An AU is a launchable item in a cmi5 course.)

The fix: use full verb IRIs, and send “initialized” first and “terminated” last.

17-tincan-answer-missing

The fault: the “answered” statement reports `result.score.scaled` as `1.5`. xAPI allows -1 to 1.

How it shows: every Tin Can milestone is reached. The error “The LRS rejected 1 statement” names `#2 result.score.scaled: result.score.scaled must be from −1 to 1`. The completion

statement, sent later, is stored.

Why the answer is missing: a real LRS refuses the statement just as Package Lab's does, so the answer is never recorded, while the completion is. Reports built from the LRS show completion and no answer.

The fix: report a scaled score between -1 and 1 (here, `1`, or the raw score with `min` and `max`).

18-lost-in-some-lmss

The fault: `scorm.js` searches for the API on its own window and one parent only (its loop stops at `i < 2`, where the sample's searches 20). An LMS that puts the course a frame deeper is out of its reach.

How it shows: fine with the generic profile, whose course sits directly under the LMS page. The **lms** experiment shows **moodle**, **scorm-cloud** and **canvas** getting only as far as **Looked for the API**: those presets put one frame between the LMS and the course. Their runs say “The content looked for API and found nothing: It looked in frames where the LMS hadn't put the API”. The **frames** experiment agrees: depth 1 and 2 fail, a new window works (the course also checks its opener).

Storage isn't it: the **scorm-cloud** profile also blocks third-party cookies, so its row could be blamed on storage. Package Lab checks: its finding, “lms scorm-cloud gets less far than the default”, says the variant blocks storage but the content never used any, so another of its settings is the cause. Moodle and canvas, which block nothing, fail the same way. The one setting they share is the frame depth.

The fix: search every parent up to the top window, then the opener, as the SCORM standard's sample code does.

19-forgets-its-answers

The fault: on its first page the course writes 5,000 characters of suspend data. SCORM 1.2 allows 4,096.

How it shows: the Problem “Suspend data is 5,000 characters”, and the warning “1 API call failed”, naming `LMSSetValue(cmi.suspend_data, …) → error 405`. The **API** view shows that call returning `false`. The location (the page) is a separate value, and it saves.

The limit: the standard's, which Package Lab's test LMS enforces. An LMS may set its own; a profile's **Suspend data 1.2** and **2004** fields change it. Some LMSs cut the data off silently instead of refusing it, which is worse: the course reads back broken data.

The fixes: for the author, save less (or compress it); for the export, publish as SCORM 2004, where an LMS must accept at least 64,000. That is a bigger change than it sounds: a different API, manifest and runtime, so check that your LMS supports 2004 and test the whole course again.

Other exercises

Short answers to the exercises not covered above. Your numbers may differ slightly.

Chapter 5. (1) A run you leave with **Save and exit** at page 3 reaches every milestone, and has no note about exit: **Save and exit** sets it to `suspend`. (2) The timed run adds "The content never called LMSFinish" and "The content never set exit", because nobody left. (3) The score is `cmi.core.score.raw`: 100 with every answer right.

Chapter 6. (1) `cmi.core.exit`, set to `suspend` (by **Save and exit** before the quiz is passed; after it, to an empty value). (2) `scorm.js` looks on the course's own window first, then its parent, where it finds the API: the lookups name `find` in `scorm.js`. (3) `lesson.js` is loaded with `defer`, so **Script order** runs it after the page's normal scripts. (4) Lines for wrong answers, or for the branch you didn't take (passed or failed), are red. (5) The quiz submission writes the interactions, the score and the status. (6) The course runs in a frame of Package Lab's own LMS page, as it would in an LMS.

Chapter 7, exercise 3. (2) `div#loading.spinner`. (4) `js/next.js`, loaded by `later()` in `js/boot.js`, which runs only after the JSON is parsed, and the parse threw. (5) The error comes just after the request for `data/config.json` returns.

Chapter 8, experiments. (1) **api both** and **api 1.2** reach **Committed**; **api 2004** and **api none** make no difference. (2) On `01-works.zip`, **api 2004** and **api none** get less far, as they should (the course uses 1.2). Depth 1, 2 and 3 and a new window make no difference: `scorm.js` searches up to 20 parents and the opener.

Chapter 11, two LMS behaviours. (1) With the API 1 second late, the course stops at **Looked for the API** and says "No LMS found": it looked once, at once. A sturdier course retries for a few seconds. (2) With commits failing, it misses **Committed**, with "1 API call failed: LMSCommit() → error 101". The course doesn't check the result, so the learner isn't told, and in a real LMS their progress since the last good save would be lost.

Chapter 12, comparing two runs. You should see the comparison list branches and functions that ran in one run and not the other: in **R**, `start` and the API calls it makes; in **F**, the `if (window.startOK)` decision going the other way. It points at the last line of `lesson.js`, as in chapter 8.

Chapter 13. (1) **Make the manifest's SCORM version match the API the scripts use**; the result tracks. (2) The capitals fix renamed the file because the only reference spells it `app.js`. It isn't fixed: the second fault (the script runs before the body exists) and the missing `vendor/missing.js` need a person. (3) Nothing to change: the trailing comma in the JSON needs a decision, so correct it in **Files** and export. Comparing `01-works.zip` with `02-blank-page.zip` shows `lesson.html` changed, `js/App.js` added, and the new findings about the missing file and the capitals: enough for the diagnosis. The folder check passes most packages, most of them broken.

Chapter 14. `batch --run` adds the furthest milestone of each package, which catches the packages that stop short (such as `06` and `13`) even though their files pass.

Appendix B: Quick reference

The guide on one page: keep it beside you while you work.

The method

1. Get the symptom in the learner's words: where, when, what they see.
2. Read the static findings, as predictions.
3. Reproduce it: run with the defaults, then move the environment towards the learner's.
4. Locate it: the first missed milestone, the first problem in loading order, the first error in time order.
5. Explain it: "it stops at *X* because *Y*, shown by *Z*".
6. Prove it: change one thing, and show it gets further.
7. Hand it on: what happens, where it stops, the evidence, the change and its proof.

The SCORM milestones

#	Milestone	If it's the first cross, look at
1	Launch page loaded	The manifest's launch path; Network
2	Scripts loaded	Network; Boot path
3	Content shown	The blank-page or loading-screen finding; Errors; Boot path
4	Looked for the API	Errors; the **waiting for** finding; Package's mode switches
5	Found the API	Lookups; **API: Both**; the **frames** experiment
6	Initialised	API (the call's result and error code)
7	Read saved data	API
8	Wrote data	API; the **waiting for** finding
9	Committed	API; the profile's **Commit**
10	Finished	Often nothing: use the course and leave it yourself

cmi5 runs: launch page loaded, scripts loaded, content shown, fetched the auth token, initialized, completed/passed/ failed, terminated. Tin Can runs: the first three, sent a statement, sent a completion statement.

Symptom to first move

Symptom	First move
Blank page	Run; the blank-page finding's evidence, in order
Stuck loading	Run; the loading-screen finding; Errors
Not tracked	Run; Lookups; **API: Both**
Doesn't resume	Record a macro; **Test saving**
Fails in Safari or on an iPad	**WebKit** + **iPad**; **Storage: Blocked**
Fails in one LMS	The **lms** experiment
Worked in the authoring tool	Network for `/…` requests; **Server needs**
Nothing fails, nothing happens	The **waiting for** finding; **Drill-down**
cmi5 or xAPI not recorded	The **xAPI** view: rejected statements and their problems

Ways to test a theory

Tool	Changes	Where
Launch settings	How the LMS hosts the course	Inspector › Settings
Profile	All of that, as a particular LMS	Settings › LMS › Profile
Changed file	The course's own code	Files › **Save as override**
Pin	A value the course reads	Settings › Advanced › Pins
Answer	A server's response	Network › **Answer this request…**
Experiment	One setting, every value	Experiments › **Run all**

Keyboard shortcuts

Keys	Does
⌘O	Open a package
⌘R	Run
⌘.	Stop
⌥⌘R	Record a macro
⇧⌘R	Test saving

Keys	Does
⌘E	Export corrected package
⌥⌘I	Show or hide the inspector
⌘1 to ⌘5	Jump to a view group
↑ ↓	Move through the views
⇧⌘H	Help
⇧⌘G	User Guide
⌘,	Package Lab's settings

On Windows, Ctrl replaces ⌘ and Alt replaces ⌥.

The commands

These need the Package Lab source code (chapter 14).

Command	Does
`package-lab doctor`	Checks this computer
`package-lab analyze course.zip`	Static findings
`package-lab run course.zip --wait 20 --out run.json`	Runs and saves
`package-lab report course.zip --run --redact --out report.html`	A report without the course content
`package-lab experiment course.zip --preset lms`	Every LMS profile
`package-lab discover course.zip`	Server needs
`package-lab drill course.zip`	Drill-down
`package-lab cycle course.zip --macro macro.json`	Test saving
`package-lab fix course.zip --out fixed.zip`	A corrected package
`package-lab diff a.zip b.zip`	Compare two packages
`package-lab batch folder --run`	Check and run a folder
`package-lab macro show macro.json`	Print a macro's steps
`package-lab serve`	The web interface

SCORM at a glance

	SCORM 1.2	SCORM 2004
API object	`API`	`API_1484_11`
Start, end	`LMSInitialize`, `LMSFinish`	`Initialize`, `Terminate`
Bookmark	`cmi.core.lesson_location`	`cmi.location`
Resume?	`cmi.core.entry`	`cmi.entry`
Left early	`cmi.core.exit` = `suspend`	`cmi.exit` = `suspend`
Suspend data limit	4,096 characters	At least 64,000 characters (3rd and 4th editions)

Appendix C: Troubleshooting Package Lab

When Package Lab itself, rather than the course, seems to be the problem. If you have the source, start with `package-lab doctor`: it checks Node, Electron, WebKit, the data folder and the web interface's port, and ends with a **To do** list. Rows that mention `npm` or the Package Lab folder are for the source only (chapter 3).

Installing and starting

Symptom	Cause and cure
The Mac says it can't open Package Lab, or can't check it for malicious software	The app isn't signed by Apple. Go to **System Settings › Privacy & Security** and choose **Open Anyway** beside the message about Package Lab, once
Windows warns about an unknown publisher	The app isn't code-signed. Choose **More info**, then **Run anyway**
`package-lab: command not found`	The command line comes with the source, not the downloaded app. With the source, the command isn't linked yet. In the Package Lab folder, run `npm link`, or use `node bin/package-lab.js` instead
`node: command not found`, or `doctor` says Node is too old	Install Node.js 20 or later from nodejs.org
“Electron couldn't be found”	Run `npm install` in the Package Lab folder, or install the app with `npm run install-app`
Opening a second package doesn't open a second window	That's by design: one window at a time. The new package goes to the window that's open
Package Lab isn't in Finder's **Open With** menu	Move the app into **Applications** and open it once. From the source, run `npm run install-app`, which builds the app and registers it. On Windows Package Lab never registers for `.zip` files: use **Open** or drop the file on the window

Opening a package

Symptom	Cause and cure
“Only .zip packages can be opened”	You dropped another kind of file. Package Lab opens ZIP files only; zip the course folder first

Symptom	Cause and cure
Nothing happens when you choose **Run**	Read the status line. A package with no readable manifest (or `cmi5.xml` or `tincan.xml`) can't be launched: its static findings say why

Running

Symptom	Cause and cure
The course window is blank and there are no findings	The run hasn't been analysed yet. Choose **Stop**: the findings come after
A run surprises you: other settings, clicks you didn't make, answers or pins	Look at the summary chip beside **Run**, the **Macro** segment, and **Pins** and **Answers** under **Settings › Advanced**. Something from an earlier test is still set
The course is very slow, or there are thousands of caught errors	Turn off **Record caught errors** in **Settings › Recording**. Tracing slows a course down too
“Playwright isn't installed” when you choose WebKit	In the source folder: `npm install`, then `make webkit` (chapter 14)
“WebKit runs need the Package Lab source checkout”	The downloaded app can't run WebKit. Use Chromium with **Storage: Blocked (Safari-like)** and **Device size: iPad**, or run Package Lab from the source (chapter 14). That

stands in for Safari's storage and screen size only, not for differences in the engine itself, such as `navigator.userAgentData` |

| The macro didn't play | Macros need Chromium. Check **Browser**, and that the macro segment isn't empty |

| The course needs a sign-in or licence server | Its requests show in **Network**. If they fail, that's a finding in itself: the course depends on a server. Use **Server needs**, or record a working copy where it can sign in |

| **Trace the code** has no effect, or the code view is empty | The run was in WebKit, which can't be traced, or the script is over 2 MB, built with `eval`, or from another site |

The User Guide

Symptom	Cause and cure
Practice package names aren't buttons, or a button says the package isn't there	Choose **Build practice kit** under the guide's contents first. Names are buttons only in the guide's own window or tab, not on GitHub or in the one-file copy
Choosing a package name seems to do nothing	It opened in the Package Lab window, which may be behind the guide. In the web interface, it opened in the other tab
The guide opens at the wrong place	It reopens where you last read. Choose a chapter in the contents
Your ticks have gone	They're kept per browser (or for the desktop app's guide window): another browser, or cleared site data, starts again

The web interface

Symptom	Cause and cure
A “401” page	Open the address with its `?token=…`, as `serve` printed it
“address already in use”	Something else is on the port. Change it in **Package Lab › Settings...**, or start with `--port`
“Lost the connection”	The server stopped or the network dropped. The page reconnects by itself
The Live picture doesn't change	It's sent only when the course changes. Click something

Data and privacy

- Package Lab's data is in `~/Library/Application Support/Package Lab` on a Mac, and `%APPDATA%\Package Lab` on Windows (**Help › Show Data Folder** opens it). Its `runs/` folder can grow large; delete old runs you don't need.
- Saved runs hold the course's scripts and stored values. Keep them as you would the course.
- Package Lab never sends course content anywhere. The course's own requests to other sites go out as they would in any browser, and **Network** shows them.

Still stuck

The Help covers every view and setting, and the getting started troubleshooting has the same basics in brief.

Glossary

Answer. A response you give to a request the content makes, in place of the server's. See chapter 8.

API. The JavaScript object an LMS puts on one of its windows for a SCORM course to talk to: `window.API` in SCORM 1.2, `window.API_1484_11` in SCORM 2004. See chapter 2.

Asset. An item that doesn't track the learner, such as a glossary page.

AU (assignable unit). cmi5's name for a launchable part of a course.

Banner. The verdict at the top of **Findings**: one word (such as *Works* or *Warning*) and a sentence saying how far the run got.

Boot path. How the launch page should load, worked out from the files: every file in order, with what loads it and whether it's there.

Caught error. An error the content handled itself. Easy to miss, because the course carries on, often in a mode that doesn't talk to the LMS.

cmi5. A set of rules on top of xAPI for launching courses from an LMS and the statements they must send. A cmi5 package has a `cmi5.xml`.

Commit. `LMSCommit` or `Commit`: asks the LMS to save what the course has written so far.

Content origin. Whether the LMS serves course files from its own domain or another one. See **Origin**.

Coordinates bar. The line under the Live view's picture: the pointer's position in pixels and as fractions of the window, and the element under it.

Coverage. Which lines of a script ran in a run. Green ran, red didn't, amber partly.

Data model. The named values a SCORM course reads and writes, such as `cmi.core.lesson_location`.

Drill-down. A Package Lab tool that explains why a run stopped as a chain of decisions, each the reason for the one before.

Entry. `cmi.core.entry` or `cmi.entry`: `ab-initio` on a first launch, `resume` when the LMS offers to resume.

Execution view. The traced events of a run: functions called and decisions made, with their values.

Exit. `cmi.core.exit` or `cmi.exit`: how the learner left. `suspend` means “coming back”, and many LMSs offer a resume only after it.

Experiment. Several runs of one item with one setting varied each time, compared in a table.

Finding. Package Lab's conclusion about a package or a run: a severity, a title, an explanation, evidence and what to try.

Folder check. File › Check a Folder...: the static findings for every package in a folder at once, and optionally a run of each. `batch` on the command line.

Frame. A page shown inside another page. LMSs usually show courses in frames, sometimes several deep.

Frame depth. How many frames sit between the LMS's window and the course.

Help. Package Lab's reference: what each view, button and setting does, in the inspector's Help panel. For while you work, where this User Guide is for learning.

Inspector. The panel at the right of the window (⌥⌘I), with **Settings**, **Help** and **Macro**.

Item. An entry in a package's structure that an LMS can launch.

Launch page. The page an item opens first: the resource's `href` in the manifest.

Live view. The course window shown inside Package Lab (or the web interface), which you can click and type into.

LMS (learning management system). The system that gives courses to learners and records their progress.

Lookup. A time the content read `window.API` or `window.API_1484_11`, recorded with whether it found one.

LRS (learning record store). Where xAPI statements are sent and kept.

Macro. A list of clicks, keys, typed text and waits that Package Lab plays into a course during a run. See chapter 9.

Macro file. A macro saved as JSON, with its events and the window size it was recorded at. See chapter 9.

Manifest. `imsmanifest.xml`: a SCORM package's description of itself, its items and its files.

Milestone. One of the steps a healthy run passes, from **Launch page loaded** to **Finished**. The first one missed is where to start looking.

Mode switch. A name in a course's code, such as `previewMode`, that decides whether it behaves as if it's in an LMS.

Options file. A JSON file of launch settings, pins, answers and changed files, for repeating a run on the command line. See chapter 14.

Origin. The scheme, host and port a page comes from, such as `https://lms.example.com`. Pages from different origins can't reach into each other's windows.

Override (changed file). Your edited copy of a file of the package, used in runs in place of the original.

Package. The ZIP file a course is delivered in.

Pin. A value Package Lab forces the content to read, from before its own code runs.

Profile. An imitation of the way a particular LMS hosts courses: origin, frames, sandbox, launch, API placement, headers, commits and limits.

Redacted report. An HTML report with the course content taken out, for sending to a vendor. Read it before you send it: file paths and parts of error messages stay. See chapter 13.

Resume. Starting a course again from where the learner left, using the bookmark and suspend data the LMS kept.

Run. One launch of an item, from **Run** to **Stop**, recorded and analysed.

Run finding. A finding from what happened in a run.

Sandbox. An attribute on a frame that limits what its page can do.

SCO (sharable content object). An item that tracks the learner through the SCORM API.

SCORM. The most widely used standard for packaging e-learning and tracking it in an LMS, in two versions: 1.2 and

1. SCORM 2004 has four editions; most LMSs and tools use the 3rd or 4th, and this guide's 2004 details are

from those.

Server needs. A Package Lab tool that works out what the content wants from its server, and a stand-in response.

Statement. An xAPI record of something a learner did: an actor, a verb and an object, such as “Learner, Test completed Fire Safety”.

Static finding. A finding from the package's files alone, before anything runs. A prediction, to be confirmed by a run.

Stop after. The setting that ends a run: *I stop it* (the default), or a time from 10 s to 5 min, after which Package Lab stops and analyses the run.

Storage. The setting that imitates a browser's privacy rules for a course in a frame: *Normal*, *Blocked* or *Partitioned*. See chapter 11.

Summary chip. The label beside **Run** listing every setting that isn't at its default.

Suspend data. `cmi.suspend_data`: the course's own saved state. Limited to 4,096 characters in SCORM 1.2 and at least 64,000 in 2004.

Test saving. Package Lab's save-and-resume test: a run that leaves the way a learner does, then a run that resumes.

Timeline. The view that draws a run's requests, API calls, errors and milestones against time.

Tin Can. The original name of xAPI. A Tin Can package has a `tincan.xml`.

Tracing. Recording every function the content's scripts ran and every decision they made, with their values.

User Guide. This book: Package Lab taught in chapters with exercises, in a window of its own (**Help › User Guide**).

Wait. A macro step that holds the macro until some text appears, a milestone is reached, the network is quiet, or a time has passed.

WebKit. Safari's browser engine, used by every browser on iPhone and iPad.

Working copy. A recording of the course where it works (its authoring tool's preview, another LMS), to compare with a failing run.

xAPI. The Experience API: courses send statements to an LRS over the web instead of calling an API object in the page.

Change log

What changed in each edition of this guide, newest first. The guide describes **Package Lab 0.3.0**, with the fixes listed under edition 1.2. Where the app and the guide disagree, the app is right: please report it.

Edition 1.3 (25 November 2026)

New

- Chapter 3 says where to download Package Lab (the website and GitHub releases), that the Mac app is signed and notarised, and that **Help › Check for Updates...** says whether there is a newer version.

Edition 1.2 (22 November 2026)

The guide becomes a book of its own, and the app problems the last edition listed as known issues are fixed.

New

- The User Guide has its own window: **Help › User Guide** (⇧⌘G), or **User Guide** in the Help panel. Contents by part, Previous and Next, On this page, search, a **Done** box on every exercise with progress in the contents, and it reopens where you left off. The web interface has the same at `guide.html`, in a tab of its own.
- The practice kit builds from the guide's window (**Build practice kit**), on the Mac or Windows, with no Terminal. Practice package names in the text open the package in Package Lab; macro file names load the macro.
- `Package Lab User Guide.html`: the whole guide in one file, written into `dist/` and the Windows zip by every build, and by `make guide`.
- `Package Lab User Guide.pdf`: the guide as a PDF, with a cover, contents, bookmarks and working links, written beside it by every build, and by `make guide-pdf`.

Changed

- Names: this book was the *Handbook* and is now the *User Guide*. What used to be called the user guide (the reference, one task to a page) is now **Help**, and stays in the inspector's Help panel. The Help menu's **Show Docs** is now **Show Help**, and **Help Topics** opens Help's index.
- About this guide explains the guide and Help, and the guide's window. Chapters 3, 4 and 9 use the window: building the kit, opening packages and loading macros. Windows users can

now do every exercise the Mac app can.
- Appendix C has a section on the guide's window.
- A review as a new reader: the counts of faulty and sound practice packages agree everywhere; what needs the source code is said up front, with each command-line-only feature marked; “run it for 10 seconds” is defined; the run list is oldest first, as the app shows it; a fixed pipeline gate (`jq`), Windows paths and menus, suspend data limits as minimums, softer claims about Safari storage and real LRSs, a short cmi5 primer before exercise 16, and more glossary terms. The findings' **Try:** says “API set to Both”, not “the Both APIs profile”.
- Chapter 3 is about the downloaded app only, on the Mac or Windows, and its first-open warnings: no Terminal, no source code. Setting up the source code (Node.js, `npm install`, `npm link`, `doctor`, `make webkit`) moves to the start of chapter 14, the only chapter that needs it, with WebKit for chapter 11. Chapters 1 and 11, Appendices B and C say what needs the source: the command line, the web interface and WebKit runs.

Fixed in Package Lab, and so no longer known issues

- **Frame depth 3 now loads the course.** The browser refused a third LMS frame with the same address as the frames around it; each LMS frame now has its own. Chapter 8's experiment and its answer are updated.
- **Experiments no longer blame blocked storage for another setting.** A variant that blocks storage, among other changes, is put down to storage only if the course used storage. Chapter 11's exercise on `18-lost-in-some-lmss.zip` and its answer are updated.
- **“Scripts run before what they need” no longer fires for code inside a function.** A use inside a listener or a callback written on the same line runs later, so it isn't flagged. `12-waits-forever.zip` has no static findings now; chapter 12's exercise, chapter 15's traps and Appendix A are updated.

Edition 1.1 (22 November 2026)

A revision after a full review of the first edition, against the app, its reference docs and runs of every exercise.

New

- Chapter 9, *Macros: playing the learner*: recording in the toolbar and in the Live view, the coordinates bar, what a macro remembers, waits, the editor, saved macros, macro files and `package-lab macro show`.
- Six practice packages: `14-next-button-late` and `15-button-moves` (with a macro file each) for macros, `16-cmi5-never-passes` and `17-tincan-answer-missing` for xAPI, and `18-lost-in-some-lmss` and `19-forgets-its-answers` for LMS hosting. The kit now has 19 packages.

- Exercises on an API that comes late, failing commits, and comparing two local runs (**R** and **F**).
- Appendix A now answers every exercise, under *Other exercises*. Appendix B, *Quick reference*, and Appendix C, *Troubleshooting Package Lab*.
- A *Beyond this guide* section in chapter 15: interactions, SCORM 2004's two statuses, sequencing, the Windows build.
- This change log, and `book.lua`, a Pandoc filter for building the guide as one document.
- A *Known issues* section, for three problems in Package Lab the exercises ran into. All three are fixed in edition 1.2.

Changed

- Chapters 9 to 14 became 10 to 15. *Saving and resuming* (now chapter 10) uses chapter 9's macro, and says that the macro carries over from one package to the next until you clear it.
- *Exercise answers* became Appendix A.
- The three severities: the window says **Problem**, **Warning** and **Note**; the command line says error, warning and info. Chapter 5 maps them, and exercises in the window now say Problem.
- Chapter 3: the full kit, `make webkit` instead of `npm i -D playwright` (which changed `package.json`), what Windows users can and can't do, and "read the code only when an exercise says so".
- Chapter 4: which settings are remembered after you quit; the **Open** button's real label; **Open saved...** and the **R** and **F** marks.
- Chapter 6: SCORM 1.2 and 2004 error codes side by side; the API table marked as shortened; a **Frames** exercise.
- Chapter 13: the corrected-package fixes under their real checklist names, including the Xyleme fix.
- Chapter 14: more flags, `--variants`, `discover --request`, `items`, exit statuses, how to gate a pipeline on a milestone, what the web interface can't do, and two integrations since taken out of the docs.

Corrected

- Chapter 5: the quiz is three **Next**s away, not two; going back to the files-only views means opening the package again (there is no entry for the package in **Runs**).
- Chapter 13: opening a saved run in the app only reads it. Repeating it needs the same ZIP and `package-lab run --options-from`.
- Chapter 14: `--strict` fails on error findings, not on a missed milestone.
- Chapter 8: the **standard** experiment on `01-works.zip` has a stated result, and the depth 3 issue is noted.
- Appendix A: the second fault in `02-blank-page.zip` names the right file for each way of fixing the first; the answer for `08-forgets-its-place.zip` no longer contradicts itself.

- Chapter 2: the manifest extract says what it leaves out; the six healthy steps “map onto” the ten milestones.
- Chapter 1: no jargon before chapter 2 explains it.
- Folder-check lessons no longer quote a count of packages that changes with the kit.

Edition 1.0 (22 November 2026)

The first edition: fourteen chapters from foundations to expert method, exercise answers, a glossary, and a practice kit of thirteen packages built by `npm run practice`.