

No-AI Full-Stack Roadmap

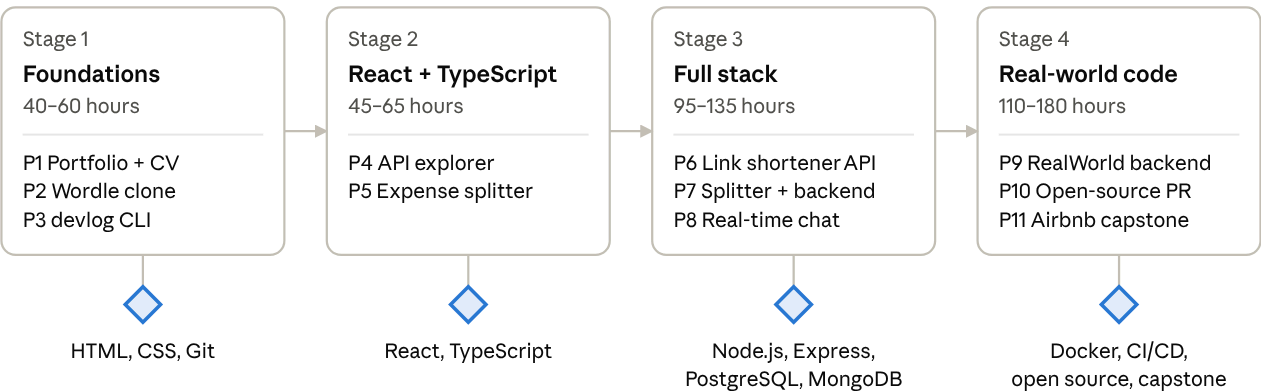
Oct 5, 2026 · @Konstantinos Bourdakos

How this works

Eleven projects in four stages take you from plain HTML to a tested, deployed full-stack app, each built without AI until it's Done. Budget about 290–440 hours: roughly 7–13 months at 8–10 hours a week. The early projects are short, so your first finish lands within two weeks.

Each stage ends with something new you can put on your CV

Diamonds mark what goes on your CV when the matching checklist boxes are ticked.



roadmap · 4 stages, 11 projects, 4 CV unlocks

Stages 3 and 4 hold about 70% of the hours, so the early wins come fast and the depth comes later.

Goal codes used below: **G1** stack to junior level · **G2** console and git · **G3** debugging and searching solo · **G4** reading and changing unfamiliar code · **G5** breaking problems down into optimal solutions · **G6** confidence to put it on your CV.

#	Project	Stack it requires	Trains	Hours
1	Portfolio + CV site	HTML, CSS	G1 G2	10–15
2	Wordle clone	TypeScript, DOM, CSS	G1 G3 G5	15–25

#	Project	Stack it requires	Trains	Hours
3	devlog CLI	Node.js, TypeScript	G2 G3 G5	15–20
4	API explorer	React, TypeScript, React Router	G1 G3	20–30
5	Expense splitter (front end)	React, TypeScript	G1 G5	25–35
6	Link shortener API	Express, PostgreSQL, TypeScript	G1 G2 G5	25–35
7	Expense splitter goes full-stack	React, Express, PostgreSQL	G1 G4 G5	40–60
8	Real-time chat	React, Express, Socket.IO, MongoDB	G1 G3	30–40
9	RealWorld: a backend for someone else's app	Express, PostgreSQL, their React code	G3 G4 G5	35–50
10	Open-source contribution	Whatever the project uses	G2 G4 G6	15–30
11	Capstone: your Airbnb clone, finished	Full stack + tests, CI, Docker	All six	60–100

Your Airbnb clone becomes P11. Bookings without double-booking are exactly the capstone's hardest problem, and returning to your own code months later is goal 4 in practice. Freeze it at a clean commit with a "where I left off" note in its README.

Rules of the game

Your two rules stay; eight additions close the gaps that usually sink plans like this: a fuzzy "finished", AI leaking in through your tools, and one bad week turning into a quit.

1. **No AI until Done.** That includes chatbots, AI autocomplete in your editor, and AI answers in search results — switch all three off before P1. Allowed: official docs, MDN, Stack Overflow, GitHub issues, blog posts, videos that explain a concept, books, and people. Not allowed: a tutorial that builds the same project.
2. **Every project requires part of the stack.** Each project lists what it requires.

3. **Done is a checklist, not a feeling.** AI comes back only when the Definition of Done below is fully ticked — as a reviewer, not an author.
4. **Plan before code.** Write the features as user stories, the data shapes, the routes or components, a task list as GitHub Issues, and an hours estimate. 20–60 minutes.
5. **Core first.** Stretch features only after Done. New ideas mid-project go into a "v2" list in the README.
6. **One roadmap project at a time.** The next one starts when the current one is Done.
7. **Rules change only between projects, in writing.** Mid-project, they're fixed.
8. **A slip isn't a reset.** If you use AI mid-project, log what you asked and why in devlog, and keep going.
9. **A minimum, not a maximum.** Three sessions a week, and 30 minutes counts. The target is 8–10 hours; the floor keeps a busy week from breaking the chain. End every session with a commit and a one-line devlog entry.
10. **Scope.** These rules cover roadmap projects only. Work with a deadline — client sites, tools you need fast — can use AI; keep it in separate repos.

Stuck? Climb the stuck protocol below before anything else.

Stack decision

Use Express for the backend and PostgreSQL as your main database; MongoDB gets one project (P8).

Express — yes. It's small, so you learn what every Node framework is built on: HTTP, routing, middleware, error handling. Use Express 5 (Node 18+): errors thrown in async handlers reach your error middleware automatically ([Express 5 guide](#)).

PostgreSQL first. It's the most-used database in the 2025 Stack Overflow survey: 55.6% of respondents, against 24% for MongoDB ([survey](#)). SQL carries over to MySQL, SQL Server, interviews and your databases course. Supabase is PostgreSQL underneath, so building your own API on it shows you what Supabase does for you.

MongoDB once. Its flexible schema makes it quick to start and easy to skip data modeling. One project is enough to list it honestly and know when it fits. If you'd rather go deeper than wider, build P8 on PostgreSQL too.

New tools arrive one or two per project, never all at once:

Tool	What it's for	First used
Vite	Dev server and build for TypeScript and React	P2
Vitest	Unit tests	P2
ESLint + Prettier	Linting and formatting	P2
tsx	Running TypeScript in Node during development	P3
React Router	Pages and state in the URL	P4
React Testing Library	Component tests	P4
zod	Runtime validation; types shared by front and back end	P5
pg (node-postgres)	Raw SQL from Node	P6
Supertest	API integration tests	P6
Prisma or Drizzle	ORM and migrations	P7
Docker Compose	Local databases, later the whole stack	P7
Socket.IO	Real-time messaging	P8
Mongoose	MongoDB models	P8
Hurl	Running the RealWorld API tests from the terminal	P9
GitHub Actions	CI on every pull request	P7 (stretch), P11
Playwright	End-to-end tests	P11

Docker is near-universal now — 71% of respondents in the same survey — which is why it appears in P7 and P11.

Stage 0 • Setup

One evening before P1: switch the AI off and set up the tools every project uses.

- ☐ Turn off AI completions and chat in your editor (Copilot or any built-in assistant)

- ☐ Turn off AI answers in your search engine, or switch to one without them
- ☐ If you're on Windows: install WSL with Ubuntu and do all roadmap work inside it — it's the same shell as the servers you'll deploy to
- ☐ Install Node with a version manager (nvm or fnm); set your git name and email; add an SSH key to GitHub
- ☐ Apply for the [GitHub Student Developer Pack](#) (free tools and credits; check what it includes now)
- ☐ Create a `roadmap` repo: these rules in the README, one checkbox per project, and an hours log
- ☐ Bookmark the docs you'll live in: [MDN](#), [react.dev](#), the [TypeScript handbook](#), [expressjs.com](#), the [PostgreSQL docs](#), the [Node.js docs](#)

Stage 1 · Foundations

Stage 1 fills the gaps React-first learners skip — layout, the DOM, Node outside the browser, the terminal — in 40–60 hours. In every project, each core feature is followed by → and the skill it trains.

P1 · Portfolio + CV site

Requires: HTML, CSS · **Trains:** G1, G2 · **Hours:** 10–15

A responsive, accessible personal site that will show every project you finish here.

Core:

- Semantic page structure: header, nav, main, sections, footer → semantic HTML
- A projects grid that goes 1 → 2 → 3 columns as the screen widens → CSS Grid, mobile-first media queries
- Light and dark themes from CSS custom properties and `prefers-color-scheme` → CSS variables
- A CV page that prints cleanly onto one page → `@media print`
- A contact form with native validation; no backend yet, P6 wires it up → forms

Git and console: create files and folders from the terminal; `init`, `add`, `commit`, `push`; a `.gitignore`; one commit per logical change; deploy with GitHub Pages.

Stretch: a mobile menu with no JavaScript; subtle CSS transitions.

Done when: live on GitHub Pages; zero errors in the W3C validator; usable at 320 px wide and by keyboard alone; Lighthouse accessibility score of 95 or more.

You can now say: "I write semantic, responsive, accessible HTML and CSS without a framework."

P2 · Wordle clone

Requires: TypeScript, DOM, CSS (Vite's vanilla TypeScript template) · **Trains:** G1, G3, G5 · **Hours:** 15–25

The word game in plain TypeScript with no framework, so later you'll know exactly what React does for you.

Core:

- Board plus on-screen keyboard, and physical keyboard input → DOM events, event delegation
- Scoring that gets duplicate letters right (a guess with two E's against an answer with one) → decomposition; solve it on paper before coding
- Vitest tests for the scoring function, including at least six duplicate-letter cases → testing pure logic
- One state object and one `render()` function; stats and an unfinished game survive a reload → state without a framework, localStorage
- Flip and shake animations triggered by toggling classes → CSS transitions

Debugging focus: step through the scoring function with a DevTools breakpoint at least once, instead of adding `console.log`.

Stretch: Greek mode, where α counts as α and ς as σ (look up `String.prototype.normalize`); share results as an emoji grid with the Clipboard API; hard mode.

Done when: tests pass; strict TypeScript with no `any`; deployed.

You can now say: "I can build an interactive app in plain TypeScript and unit-test its logic."

P3 · devlog CLI

Requires: Node.js, TypeScript · **Trains:** G2, G3, G5 · **Hours:** 15–20

A terminal tool for logging bugs, lessons and study sessions. You'll use it for the rest of the roadmap.

Core:

- `devlog add "fixed CORS" --tag cors`, `devlog list --tag cors --since 7d`, `devlog search <text>` → argument parsing; write your own small parser, no library
- `devlog session 45 "scoring tests"` and `devlog stats`: minutes this week, current streak, top tags → aggregation, and a visible record of progress
- Entries in a JSON file in your home folder; a missing or corrupted file never crashes it → `fs`, `path`, `os`, error handling
- Fuzzy search that still finds "authentication" when you type "authentification" → Levenshtein distance (dynamic programming); work out its time complexity
- Proper CLI behavior: errors to `stderr`, correct exit codes, output that pipes into `grep`; installed globally with `npm link` → how terminal tools work

Git and console: from here on, every feature gets its own branch and a pull request on GitHub, even solo. Create a merge conflict on purpose and resolve it in the terminal. Read history with `git log --oneline --graph`.

Stretch: `devlog export --md` for a weekly summary; colors with raw ANSI codes; publish to npm.

Done when: works from any folder; bad input gets a helpful message; tests cover the parser and the fuzzy matcher.

You can now say: "I'm comfortable in the terminal, with Node outside the browser, and with a branch-and-PR workflow."

Stage 2 · React + TypeScript

Stage 2 makes React and TypeScript yours — async data, routing, complex state — in 45–65 hours.

P4 · API explorer

Requires: React, TypeScript, React Router · **Trains:** G1, G3 · **Hours:** 20–30

A searchable browser for a free public API you like: books (Open Library), countries (REST Countries), or anything similar.

Core:

- Search as you type, debounced with your own `useDebounce` hook → custom hooks, effects

- Fast typing never shows stale results: reproduce the race first with network throttling, then fix it with `AbortController` → async race conditions
- Loading, error and empty states for every request; TypeScript types written from the real JSON → typing external data
- List and detail pages; search terms and filters live in the URL, so back/forward and shared links restore them → React Router, URL as state
- Favorites saved with a generic `useLocalStorage<T>` hook → generics

Debugging focus: React DevTools for props and state; the Network tab with throttling on.

Stretch: an in-memory cache so repeated searches are instant; infinite scroll with `IntersectionObserver` ; a first React Testing Library test.

Done when: no stale results under throttling; no `any` ; deployed.

You can now say: "I can build a typed React app on a real API and handle its async edge cases."

P5 · Expense splitter (front end only)

Requires: React, TypeScript · **Trains:** G1, G5 · **Hours:** 25–35

Split shared costs in a group — a trip, a flat's bills — and see who owes whom. Front end only for now; you'll come back to this code in P7.

Core:

- Groups → expenses → splits (equal, exact amounts, percentages) in a dynamic form with validation → controlled inputs, complex forms
- All state in one `useReducer` with typed actions (a discriminated union and an exhaustive `switch`) → reducers, TypeScript unions
- Money stored as integer cents: €10 split three ways is 3.34 + 3.33 + 3.33, never 9.99 → floating-point bugs, remainder handling
- Balances derived from expenses, never stored → derived state
- "Settle up" in at most $n - 1$ payments, the biggest debtor paying the biggest creditor first → greedy algorithms; in the README, explain why it's at most $n - 1$ and why the true minimum is a much harder problem
- Undo and redo for every action → stacks of past and future states
- Export and import a group as JSON, validated with zod → runtime validation

Tests: reducer and settle-up tests; one React Testing Library test of the expense form.

Stretch: several currencies with fixed rates; spending by category as an SVG chart you draw yourself.

Done when: balances always sum to zero (a test proves it); undo and redo never corrupt state; a broken import shows an error instead of crashing.

You can now say: "I can model complex state in React with TypeScript, and test the logic behind it."

Stage 3 · Backend and full stack

Stage 3 adds the server and the database, and SQL gets the most practice of anything in the roadmap — 95–135 hours.

P6 · Link shortener API

Requires: Express, PostgreSQL (raw SQL with `pg`), TypeScript · **Trains:** G1, G2, G5 · **Hours:** 25–35

A URL shortener with click stats: small, but it touches HTTP, SQL, concurrency and security.

Warm-ups first:

- A server in about 30 lines with Node's built-in `http` module, so you see what Express does for you
- An hour in `psql` with no GUI: create the tables by hand, then write about 15 queries (filters, JOINS, GROUP BY, an index)

Core:

- Create a link (with an optional custom alias), redirect from `/:code`, read stats, delete → REST routes and status codes (201, 301 vs 302, 404, 409 for a taken alias)
- Short codes: compare random codes (with collision handling) against base62-encoding the row id; pick one and justify it in the README → trade-offs
- Click counts that stay correct under load: increment in SQL (`clicks = clicks + 1`), then prove with autocannon that a read-then-write version loses clicks → race conditions, atomic updates
- zod validation, plus one error middleware that returns consistent JSON errors → middleware

- Parameterized queries only; first try a SQL injection against a deliberately unsafe copy → security basics
- Config from environment variables; `.env.example` committed, `.env` ignored → configuration

Testing: every endpoint by hand with `curl` first, then Supertest integration tests against a separate test database.

Stretch: your own rate-limiting middleware; clicks per day with `GROUP BY date_trunc(...)`; a `POST /contact` route wired to your P1 form (revisiting old code).

Done when: every endpoint has a test and a `curl` example in the README; the load test shows zero lost clicks.

You can now say: "I can design, build and test a REST API with Express and PostgreSQL."

P7 · Expense splitter goes full-stack

Requires: React, Express, PostgreSQL (Prisma or Drizzle), TypeScript · **Trains:** G1, G4, G5 · **Hours:** 40–60

A group that lives in one browser's `localStorage` can't be shared, so the app gets accounts, an API and a database.

First hour: read your P5 code cold, without running it. Write down everything that confuses you; that list is your first refactor list.

Core:

- Monorepo with npm workspaces: `client/`, `server/`, `shared/`, with zod schemas shared by both sides → one source of truth for validation
- Session auth: register, log in, log out, passwords hashed with argon2 or bcrypt, `httpOnly` cookies → auth fundamentals
- Invite friends to a group; only members can see it, and a test proves an outsider is refused → authorization
- Schema and migrations for users, groups, memberships (many-to-many), expenses and splits → relational modeling
- Balances computed in SQL, not by loading every row into JavaScript; find and fix one N+1 query by logging queries → query performance
- Paginated expense history: start with offset, and note in the README why cursor pagination exists → pagination

- PostgreSQL in Docker Compose locally; client, server and database deployed → environments

Stretch: TanStack Query with optimistic updates; `EXPLAIN ANALYZE` before and after adding an index; GitHub Actions running tests on every pull request.

Done when: two accounts can't see each other's groups (tested); it works on your phone; the README has an architecture diagram.

You can now say: "I can build and deploy a full-stack TypeScript app with auth and a relational database."

P8 · Real-time chat

Requires: React, Express, Socket.IO, MongoDB (Mongoose) · **Trains:** G1, G3 · **Hours:** 30–40

Rooms, live messages and presence — and your one MongoDB project, so you can compare it with PostgreSQL from experience.

Core:

- Rooms and live messages over Socket.IO, with session auth like P7's applied to socket connections → WebSockets, event-driven code
- History with cursor-based infinite scroll upward, backed by an index → cursor pagination, MongoDB indexes
- Typing indicators and online presence that stay correct when a tab closes or the connection drops → cleanup, reconnection edge cases
- Unread counts per room → efficient queries
- A half-page note in the README: did you embed or reference, and when would PostgreSQL be the better choice? → document vs relational modeling

Debugging focus: WebSocket frames in the Network tab; offline mode to force disconnects; two browsers side by side.

Stretch: image messages with type and size checks; reactions; deploy with a hosted MongoDB.

Done when: two browsers stay in sync through refreshes, disconnects and reconnects; scrolling back through history never duplicates or skips a message.

You can now say: "I've built real-time features and can explain when MongoDB fits and when PostgreSQL does."

Stage 4 · Other people's code, and production

Stage 4 is where goal 4 gets real training — code you didn't write, code you haven't opened in months, code strangers review — in 110–180 hours.

P9 · RealWorld: a backend for someone else's front end

Requires: Express, PostgreSQL, JWT, plus a React front end you didn't write · **Trains:** G3, G4, G5 · **Hours:** 35–50

[RealWorld](#) is an open spec for Conduit, a Medium-style blogging app with over 100 implementations. Every front end works with every spec-compliant backend — including yours.

Core:

- Pick a React front end from [CodebaseShow](#), get it running, and try it against the public API at [api.realworld.show](#) so you know what working looks like; fix whatever has rotted using changelogs and migration guides → legacy setup
- Before changing anything, map it: folder structure, where state lives, and one feature traced from click to network request → reading unfamiliar code
- Build your Express + PostgreSQL backend from the [OpenAPI spec](#): JWT auth (`Authorization: Token ...`), profiles and follows, articles with unique slugs, comments, favorites, tags, the feed → implementing someone else's contract
- Run the spec's [Hurl test suite](#) until every test passes; where the prose docs and the tests disagree, the tests win → tests as the source of truth
- Point their front end at your backend and fix the integration bugs → cross-stack debugging (CORS, auth headers, response shapes)
- Add one feature across both codebases — bookmarks, or reading time — in their code style → changing code you didn't write

Stretch: convert one part of their front end to TypeScript; write a review of their code as if it were a pull request.

Done when: the Hurl suite passes; their front end works fully against your backend; your feature works end to end.

Time-box: if a front end won't run after 4 hours, pick another implementation.

You can now say: "I can join an existing codebase, build to an API spec, and ship a feature across front and back end."

P10 · Open-source contribution

Requires: whatever the project uses (filter for TypeScript, React or Node) · **Trains:** G2, G4, G6 · **Hours:** 15–30

A real fix in a real project, reviewed by people who don't know you.

Core:

- Find an issue labeled "good first issue" in an active project (goodfirstissue.dev, up-for-grabs.net, or GitHub's own search) and read its CONTRIBUTING guide → how real teams work
- Fork, clone, and get the dev environment and test suite running → setting up an unfamiliar project
- Reproduce the issue; find the code with `grep` or `rg`, go-to-definition and `git blame` → navigating a large codebase
- Fix it, add or update a test, and open a pull request that follows their conventions; answer every review comment → real code review

Done when: your PR is open with CI passing and every review comment answered. Merged is a bonus; that part depends on the maintainers.

You can now say: "I've contributed to open source" — with a link. Docs and test fixes are valid first PRs; they teach the same workflow.

P11 · Capstone: your Airbnb clone, finished

Requires: the full stack, plus tests, CI and Docker · **Trains:** all six · **Hours:** 60–100

Your Airbnb clone comes back as the final project: booking dates without conflicts is exactly the hard problem a capstone needs. Alternative: a booking system for a service business — a clinic, a law office, a tutoring school — which doubles as a demo for client work.

First session: read your old code cold and list what past-you got wrong. Then write a two-page design doc: roles, user stories, data model (ERD), API outline, risks.

Core:

- Hosts, guests and an admin, with permissions enforced on the server → authorization beyond "logged in"

- Availability and bookings with no double-booking: write the naive overlap check, then a better one (sort and sweep, or a PostgreSQL exclusion constraint), and wrap booking in a transaction so two simultaneous requests can't both win → decomposition, optimal solutions, concurrency
- Search by dates, price and guests that stays fast on 10,000 seeded listings → seed scripts, indexes, pagination
- Image uploads with type and size validation → file handling
- Dates stored in UTC and shown in local time → time zones
- Payments in Stripe test mode, or confirmation emails through a provider's free tier → third-party APIs
- Tests at three levels: unit (overlap logic), API (Supertest), one end-to-end booking (Playwright) → the testing pyramid
- Docker Compose for the whole stack; GitHub Actions running lint, type-check and tests on every PR; deployed with logs you could debug from → CI/CD, operations

Stretch: deploy to a Linux server yourself — SSH, an Nginx reverse proxy, HTTPS with Let's Encrypt; a host dashboard with charts.

Done when: a stranger can book a stay without your help; a test fires two simultaneous bookings for the same dates and exactly one succeeds; CI is green; the README, design doc and architecture diagram are in the repo.

You can now say: "I designed, built, tested and deployed a production-style full-stack app." This is your headline CV project.

Running alongside every project

Five habits run in the background and train what single projects can't.

- **devlog (from P3):** every bug that costs you more than 15 minutes gets an entry — symptom, guesses, what you tried, root cause, fix, how you'd catch it faster next time. Until P3 exists, use a text file. → G3
- **Revisit after every third project:** one session in an old project, cold — run it, read it, fix one thing, update its dependencies. → G4
- **Code reading, an hour a week:** a small library's source, a merged pull request in a popular repo, or a codebase you built with AI help — explain one file in your own words, then change something by hand. → G4

- **Algorithms, two problems a week** in TypeScript: plan on paper, state the time complexity, then code. → G5
- **Console:** keep climbing [OverTheWire Bandit](#); use git from the command line only, never a GUI client. → G2

Git skills arrive in the order the projects force them:

Git skill	First needed in
<code>init</code> , <code>add</code> , <code>commit</code> , <code>push</code> , <code>.gitignore</code>	P1
Branches, pull requests, resolving a merge conflict	P3
<code>rebase</code> , interactive rebase to squash and reword	P4–P5
<code>stash</code> , <code>revert</code> , <code>reset</code> , <code>reflog</code> (recover a "lost" commit on purpose)	P6
<code>bisect</code> , to find the commit that broke something	P7
Forks, upstream remotes, review etiquette	P9–P10
Tags and releases	P11

Stuck protocol

Climb this ladder in order before anything else — and asking a person is always allowed.

1. Read the whole error and stack trace; find the first line that points into your own code.
2. Reproduce it on demand. Without a reproduction, you can't confirm a fix.
3. Shrink it: remove or isolate code until the smallest version still fails.
4. Look instead of guessing: breakpoints, the actual values, the Network tab, React DevTools, `psql`.
5. Write one hypothesis in devlog, test it, record the result. Repeat.
6. Read the official docs for the exact function or API involved, for the version you're using.
7. Search the exact error message in quotes, plus the library name; check the library's GitHub issues.
8. Rubber-duck it: write the question as if posting to Stack Overflow — expected, actual, what you tried. Writing it often answers it.

9. After 2 focused hours on one issue, switch to another task. Still stuck next session?
Ask a person: a classmate, a friend, a Discord community, Stack Overflow.
10. Still stuck after that: park it as a known issue in the README, move on, and come back with fresh eyes.

Definition of Done and the AI review

A project is Done when every line below is true; only then does AI come back — as a reviewer, never an author. Copy the list into each project's README and tick it there.

- Every core feature works, checked against your plan
- TypeScript strict mode with zero errors; any `any` has a comment saying why
- ESLint and Prettier pass
- The tests the project lists pass
- README: what it is, a screenshot or demo, how to run it, key decisions, known limitations
- Deployed with a live link, if it has a UI or an API
- Meaningful commit messages; from P3 on, merged through pull requests
- devlog entries for the bugs that cost you time
- A half-page retro: hardest part, estimated vs actual hours, what you'd do differently

The review loop, after Done:

1. Self-review first: read your code as if it were a stranger's pull request, and note five things you'd change.
2. Ask the AI for a review with the prompt below: critique and hints, no rewrites.
3. Compare. Whatever it caught that you missed is a blind spot; add it to your plan template for the next project.
4. Make every fix yourself, by hand.
5. Add the project to your portfolio with one CV bullet: what you built, with what, and one concrete result.

The review prompt:

I'm learning full-stack development and built this project without AI. Review it the way a senior engineer reviews a junior's pull request.

List the most important issues in order of severity: bugs, security, data handling, design, readability. For each one, say where it is (file and line), why it matters, and give a hint toward the fix.

Don't rewrite my code or give full solutions; I'll make the fixes myself. Ask me about anything where my intent isn't clear. Finally, tell me which concept to study before my next project.

Junior-ready checklist

A tool goes on your CV when every box under it is ticked — tick them as the projects prove them.

HTML and CSS (P1–P2)

- ☐ Semantic markup, and forms with native validation
- ☐ Layouts built from a design with Flexbox and Grid, mobile-first
- ☐ Accessibility basics: labels, alt text, focus states, contrast, keyboard use

TypeScript (P2–P7)

- ☐ Types for props, state, events and API responses
- ☐ Unions, narrowing and generics in my own hooks and utilities
- ☐ Strict mode without `any`; runtime validation of external data

React (P4–P7)

- ☐ Components, props, state, and derived data instead of duplicated state
- ☐ Effects, and knowing when not to use one
- ☐ Forms, routing, and data fetching with loading and error states
- ☐ Context, reducers and custom hooks; at least one tested component

Node.js and Express (P3, P6–P9)

- ☐ REST API design with correct status codes
- ☐ Middleware, central error handling, validation
- ☐ Auth with sessions and with JWT; config from environment variables
- ☐ Integration tests with Supertest

PostgreSQL (P6–P11)

- ☐ Schema design with keys and constraints; migrations
- ☐ JOINS, aggregates and parameterized queries
- ☐ Indexes checked with EXPLAIN; transactions

MongoDB (P8)

- ☐ Document modeling: when to embed and when to reference
- ☐ Queries, indexes and cursor pagination

Git and console (all projects)

- ☐ Branch, pull request, merge; resolving conflicts
- ☐ `rebase`, `revert`, `reset`, `reflog`, `bisect`
- ☐ Navigating, searching and piping in the shell; env vars; processes and ports; `curl`; `ssh`

Debugging (all projects)

- ☐ Breakpoints in the browser and in Node
- ☐ Reading stack traces; the Network tab; narrowing a bug down step by step

Shipping (P7, P11)

- ☐ Deploying a front end, a back end and a database
- ☐ CI that lints, type-checks and tests every pull request
- ☐ Docker Compose for local development

On the CV, link the proof rather than the list: the capstone, the full-stack expense splitter, your open-source PR and your portfolio.

Sources

- [2025 Stack Overflow Developer Survey — Technology](#): database and Docker usage
- [Upgrade to Express v5](#): async error handling, Node 18+
- [RealWorld on GitHub](#): the spec, the implementations, the public API
- [RealWorld backend spec: introduction](#): OpenAPI spec and Hurl test suite
- [RealWorld backend spec: endpoints](#): JWT header and endpoint list