

Introduction — Full-Stack Web Development

The Course, the One App, and the Spine

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract Orientation for the course: we anchor everything to a single working full-stack app, `fullstack-minimal-app` — a React + Vite + Tailwind frontend, a Node.js/Express REST backend, and a MySQL database. The 18-hour spine is understand → extend layer-by-layer → capstone (the review layer); a compact web-architecture refresher and a Docker Compose / devcontainer environment round out the session.

Table of contents

1	About the Lecturer	1
2	Course Overview	1
	2.1 Learning objectives ☒	2
	2.2 Evaluation	2
3	The One App	2
4	The Spine	3
5	Git in Brief	4
6	Web-Architecture Refresher	5
	6.1 Front-end, back-end, full-stack	5
	6.2 Client-server	5
	6.3 Single-Page Application (SPA)	6
7	Dev Environment	7
	7.1 Get the app	7
	7.2 Services	7
	7.3 Bring it up	7
	7.4 Where the catalog renders	7

1 About the Lecturer

Emmanuel BRUNO — Associate Professor, University of Toulon (Information System Laboratory, LIS — UMR CNRS 7020).

Contact: course Discord channel (preferred for quick questions) · email emmanuel.bruno@univ-tln.fr.

2 Course Overview

One app, extended layer by layer, into a full review layer.

Goal. By the end of the course you can build and reason about a complete full-stack web application — a modern React frontend talking to a Node.js/Express REST API backed by MySQL — and you can keep extending it.

Audience. L3 / 3rd-year undergraduate students. A programming background is assumed. If you come from PHP, your server-side in-

instincts (routing, sessions, database queries) transfer directly; we apply them in the JavaScript ecosystem.

Shape: 18 hours = 6 sessions × 3 h. Each session is a mix of short lecture and hands-on work, always on the *same* application.

!🛠️ AI-first, Day 1

We start with an AI assistant (e.g. GitHub Copilot, Cursor, or similar) and LLMs before refreshing the fundamentals. They accelerate learning and mirror how professional teams work — but they **never replace understanding the code**. You will always be able to explain what you shipped.

2.1 Learning objectives ☒

- **Build a complete full-stack app** with React (frontend), Node.js/Express (backend), and MySQL (persistence).
- **Design and consume a RESTful API** that connects the frontend to the backend and the database.
- **Apply client-server architecture** to a working product catalog you extend yourself.
- **Use AI tools effectively** to develop faster while understanding every line you generate.

2.2 Evaluation

- **Final quiz** (individual, 30 minutes at the start of Session 5, before capstone wiring).
- **Team project** (group): code + [README.md](#) + [LLM_Usage.md](#) + video demo — due 1 month after the last session.

3 The One App

💡 The one-app spine

`fullstack-minimal-app` is the **only** running example in this course. We do not build a toy app and throw it away, then start another. We take one real, working application and grow it. Understand it → extend it layer by layer → capstone: build the review layer.

What it is. A minimal but complete, production-minded full-stack application:

```
Frontend (React + Vite + Tailwind)
  ↓ HTTP/JSON
Backend (Node.js + Express)
  ↓ SQL
Database (MySQL 8.0)
```

It is deliberately small enough to read end to end, but complete enough to be real: a product catalog with categories, pagination and filtering, full CRUD, input validation, SQL-injection protection, health monitoring, and Docker-first development.

The working catalog you will extend. You start from code that already runs. The frontend is a React single-page app; its source is small and fully inspectable:

- `frontend/src/main.jsx` — the React entry point (mounts the app).
- `frontend/src/App.jsx` — the main component: client-side routing and a backend health check.
- `frontend/src/components/ProductList.jsx` — the paginated product listing.
- `frontend/src/components/ProductDetail.jsx` — the single-product view.

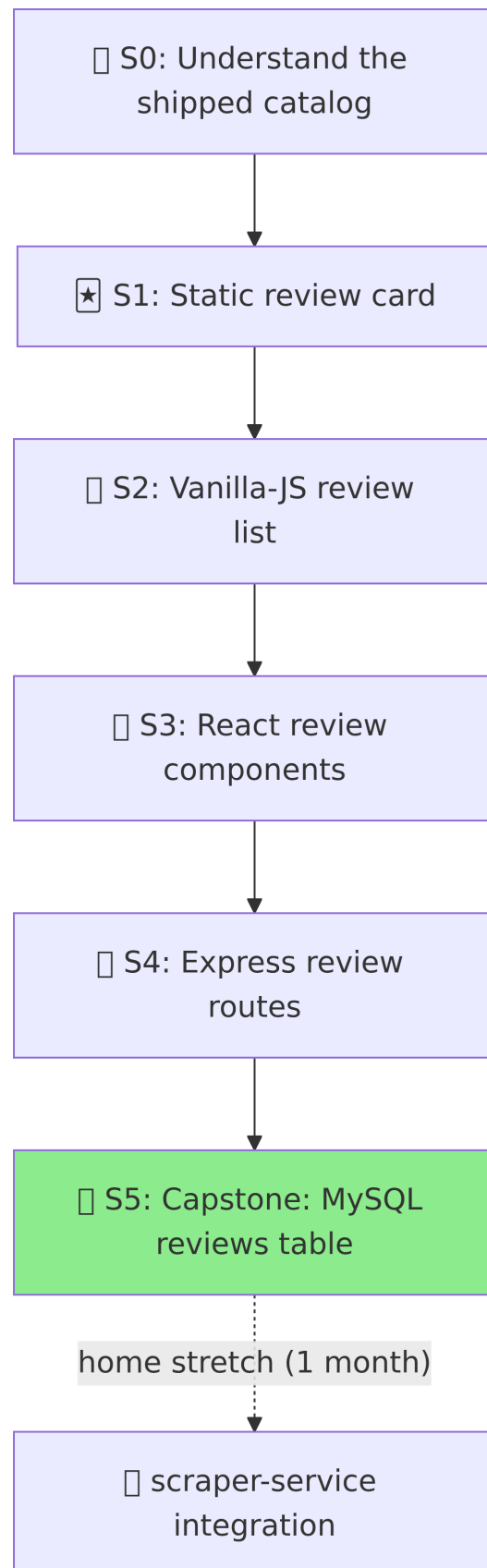
- `frontend/src/components/ProductForm.jsx` — the create / edit form.
- `frontend/src/styles.css` — Tailwind imports.

The backend exposes a REST API (`/api/products` with `GET/POST/PUT/DELETE`, `/api/categories`, and a `/health` endpoint) that reads and writes a MySQL database (`minimal_app_db`, seeded by `db/init.sql`).

Your job is to extend this catalog — never to rebuild it from scratch. Every lesson adds one layer on top of what is already working.

4 The Spine

The course is one arc, in one order: **understand the shipped catalog → add the review layer session by session → capstone.** The product catalog is the *given* foundation — you never rebuild it. **The course adds a review layer to an existing product catalog**, milestone by milestone (S0–S5); connecting that layer to the external review source (the shipped `scraper-service`) is the **home stretch**, done after the course — never wired inside the 18 hours.



Session	What you build — the review layer on the shipped catalog	Session	What you build — the review layer on the shipped catalog
S0	Understand the shipped catalog — the working app and its dev environment (this lecture).		for the shipped <code>scraper-service</code> (connected in the home stretch).
S1	Static review card — one review, presented by hand in HTML/CSS.	 The home stretch is not in the 18 h Wiring the shipped scraper service into a multi-source review aggregator is the one-month home stretch you do after the course, as part of the team project. During the 18 hours you build the review layer yourself; in the home stretch you connect it to the <code>scraper-service/</code> shipped with <code>fullstack-minimal-app</code>.	
S2	Vanilla-JS review list — render and filter reviews from a local JSON file.		
S3	React review UI — <code>ReviewCard</code> , <code>ReviewList</code> and <code>ReviewForm</code> built in React, fed with sample data.		
S4	Express review routes — <code>GET /api/products/:id/reviews</code> (plus the aggregate endpoint) over an in-memory store.		
S5	Capstone: the MySQL reviews table — full front-end → API → DB wiring, the integration seam		

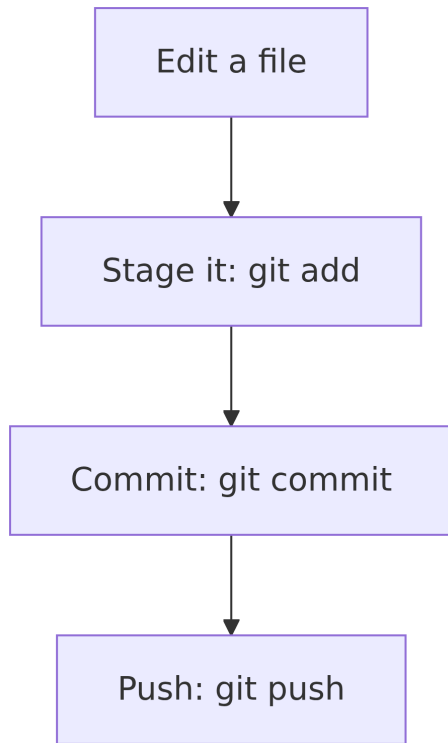
5 Git in Brief

You will work in a git repository for the whole course. Git — a version control system — is the safety net: every piece of work you commit can be inspected, reverted, or restored, and it doubles as a record of how the app grew. You do not need to master git to take the course, but you do need to be comfortable with the small daily loop below.

- **Repository (repo)** — the project folder tracked by git; here, the course app `fullstack-minimal-app`.
- **Commit** — a dated, named snapshot of the files you choose to save; the atomic unit of history.
- **Branch** — an independent line of work you can extend without disturbing the main line.

The daily loop is **edit** → **stage** → **commit** → **push**: you change files, mark the ones you want to keep (`git add`), record them as a

commit, and push the commit to the remote repository so it is backed up and shared.



How this course uses GitHub. The course lives in a single repository. At the end of each session, the work you produced is committed to the course repo — your progress is the commit history, not a pile of loose files. This also closes the AI-first loop: the **generate** → **verify** → **integrate** cycle always *ends* with a commit, so AI-assisted code enters the app only once it is reviewed and checked in.

6 Web-Architecture Refresher

A compact, mandatory refresher — just enough to place every concept we will use.

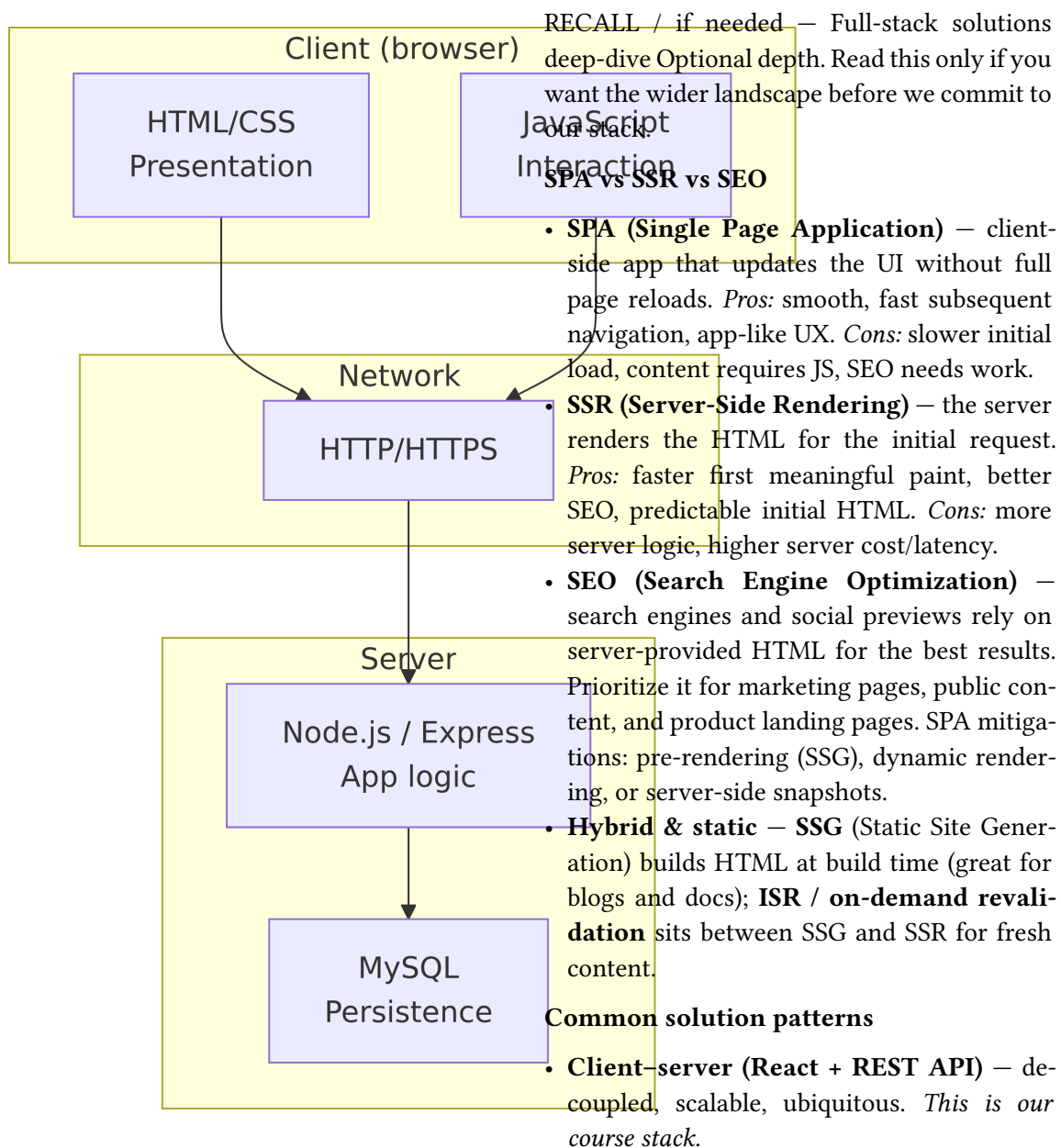
6.1 Front-end, back-end, full-stack

- **Front-end** — the code that runs in the user's browser: UI, interactions, client-side state. Here: React + Vite + Tailwind.

- **Back-end** — the server-side logic: business rules, data access, security, and the API. Here: Node.js + Express + MySQL.
- **Full-stack** — building (and understanding) both sides, and the connection between them.

6.2 Client-server

A web app is a client (the browser) talking to a server over HTTP. The client renders and collects input; the server validates, runs the business logic, and persists data.



6.3 Single-Page Application (SPA)

A SPA loads one HTML shell and then updates the UI in place — without full page reloads — by fetching data (JSON) from an API. Our catalog is an SPA: `App.jsx` sets up the routes, and components like `ProductList` and `ProductDetail` swap in as you navigate. SPAs give a smooth, app-like experience; the trade-off is that first load needs JavaScript, which is why SEO matters for public pages.

- **Monolith (Express + templating)** — simple for small projects.

Choosing a stack (quick guideline)

- SEO + fast first paint → SSR/hybrid (Next.js-style).
- Large, strictly-structured enterprise app → Angular.
- Rapid API-driven SPA → **React + Express (ours)**.
- Small / simple site → server-rendered pages (templating).

Our minimal full-stack stack

- Frontend: React + Vite (Tailwind for styling).
- Backend: Node.js + Express.
- Database: MySQL (structured data).

Why this stack for learning. One language (JavaScript) across front and back reduces context-switching; it is JSON-native and API-first; and it maps to a large job market. Once you understand React + Express + a relational database, moving to TypeScript, SSR/hybrid frameworks, or a different backend is mostly translation.

7 Dev Environment

We develop inside a **devcontainer** — a Docker container that gives everyone the same tooling — and we run the app with **Docker Compose**. The compose file starts the services the app needs; for daily development you open the project in the container and use `npm run dev`.

7.1 Get the app

```
git clone https://github.com/ebpro/
fullstack-minimal-app.git
cd fullstack-minimal-app
```

Clone the default branch; do **not** add `-b develop` to the command.

7.2 Services

Service	What it is	Host port
<code>mysql</code>	MySQL 8.0 database (<code>minimal_app_db</code> , seeded by <code>db/init.sql</code>)	3306
<code>backend</code>	Express REST API (Node.js)	4000
<code>frontend</code>	React + Vite dev server	5173
<code>scraper</code>	Mock external review API (home stretch only — use the shipped service)	5000

7.3 Bring it up

- **Quick check (no setup):** from the app root, `docker compose up --build`.
- **Daily development (recommended):** open the project in VS Code → *Reopen in Container* (accept the prompt) → in the terminal run `npm install` (if needed) then `npm run dev`. Hot reload is on: save a file, see the change.

7.4 Where the catalog renders

- **Frontend (the catalog):** `http://localhost:5173` — this is where you will see and extend the UI.
- **Backend API:** `http://localhost:4000` (health check at `http://localhost:4000/health`).

- **Scraper (home stretch only):** `http://localhost:5000`.

💡 Two common hiccups

- **Frontend shows no content:** inside the container, Vite must bind to `0.0.0.0`, not `localhost`. Check `VITE_HOST=0.0.0.0` in `frontend/.env`.
- **Port already in use:** free ports `5173`, `4000`, or `3306` (stop a local MySQL or other container) or change the mappings in `docker-compose.yml`.