

Modern Web Stack

Reading time: ~10 min · Prereqs: Lessons 01–03 · *Snapshot 2026 — framework names move; the three-tier shape doesn't.*

Why this matters to you

Almost every product you touch online — Gmail, your bank's website, your company's HR portal, the shop where you bought your last pair of sneakers — shares the same basic shape: a browser talking to a server talking to a database. Once you can see that shape, you can ask the right questions: *what's slow?*, *where does the data live?*, *what changed when?*, *who owns this piece if it breaks?* This lesson hands you the shape and the vocabulary. After it, the rest of Module 2 — programming (Lesson 05), architecture (Lesson 06), and databases (Lesson 07) — all slot neatly in. You don't need to write code to use this vocabulary; you need it to read a job posting, sit in a planning meeting, or scope a vendor.

1. What "web app" means in 2026

A **web app** is a website that *does* something rather than just *shows* something. Webmail (Gmail, Proton Mail). Collaborative documents (Google Docs, Notion). Streaming (Spotify, Netflix). Online banking. Booking tools. Your company's Jira. The line between "web app" and "desktop app" has blurred — many apps you double-click (Slack, VS Code, Discord, Notion desktop) are literally websites wrapped in a desktop window using a framework called **Electron**. Mobile is similar: a huge share of the apps on your phone are either thin shells around a web experience or native apps talking to the same backend that the web version uses.

One mental model fits almost all of them: **three tiers**. Once you can name the tiers, you can read any product like a map.

2. The three-tier model

Modern web apps split into three layers, each with one job.

- **Tier 1 — the client.** Whatever's on the user's device: a browser tab, a mobile app, sometimes a desktop window. Its job is to draw the screen and capture clicks and typing.
- **Tier 2 — the server.** Code running in a data center, listening for requests. Its job is to apply the rules (who's allowed to do what?) and to mediate every conversation with the database.
- **Tier 3 — the database.** Where state lives long-term: your account, your messages, your photos, your order history. Its job is to remember.

The client doesn't talk to the database directly — it talks to the server, and the server is the only thing trusted to read or write the database. This separation isn't bureaucracy; it's how you prevent a malicious or buggy client from running `DELETE * FROM users;`. Walk through one click: you press *Send* in Gmail → your browser sends an HTTP request to Google's mail server → the server checks your auth, writes a row, fans the message out to recipients' inboxes (more database writes) → returns success → your browser updates the screen. That tiny dance happens millions of times a second across the internet.

3. Frontend — what runs in your browser

The **frontend** is everything that runs on the client. At the bottom it's just three languages, all of which your browser speaks natively:

- **HTML** — the structure (headings, paragraphs, buttons).
- **CSS** — the style (colors, spacing, fonts, layout).
- **JavaScript** — the behavior (what happens when you click, type, or scroll).

For anything beyond a brochure site, a **framework** sits on top of those three languages to keep big UIs sane. As of 2026 the most common are **React** (made at Meta, biggest mindshare and largest job market), **Vue**, **Svelte**, and **Angular** (Google, enterprise-leaning). **Next.js** is "React plus a backend bundled together" and is currently the default new-project choice for many teams. A modern frontend ships somewhere between 100 KB and 1 MB of JavaScript to your browser the first time you visit a site; the browser then runs that JavaScript, the JavaScript draws the UI, and after that it chats with the server in the background as you click around.

4. Backend — what runs in the data center

A **backend** is an **HTTP server** (a program that listens for web requests) plus your **app code** (the rules of *your* product). Common language-plus-framework combinations as of 2026:

- **Python + Django** or **FastAPI**
- **Node.js + Express** or **NestJS**
- **Ruby + Rails**
- **Java + Spring Boot**
- **C# + ASP.NET**
- **Go** with its standard library (frameworks are deliberately minimal)
- **PHP + Laravel** (still huge — WordPress, Wikipedia, much of Facebook's origin)
- **Elixir + Phoenix** (smaller, beloved for real-time apps)

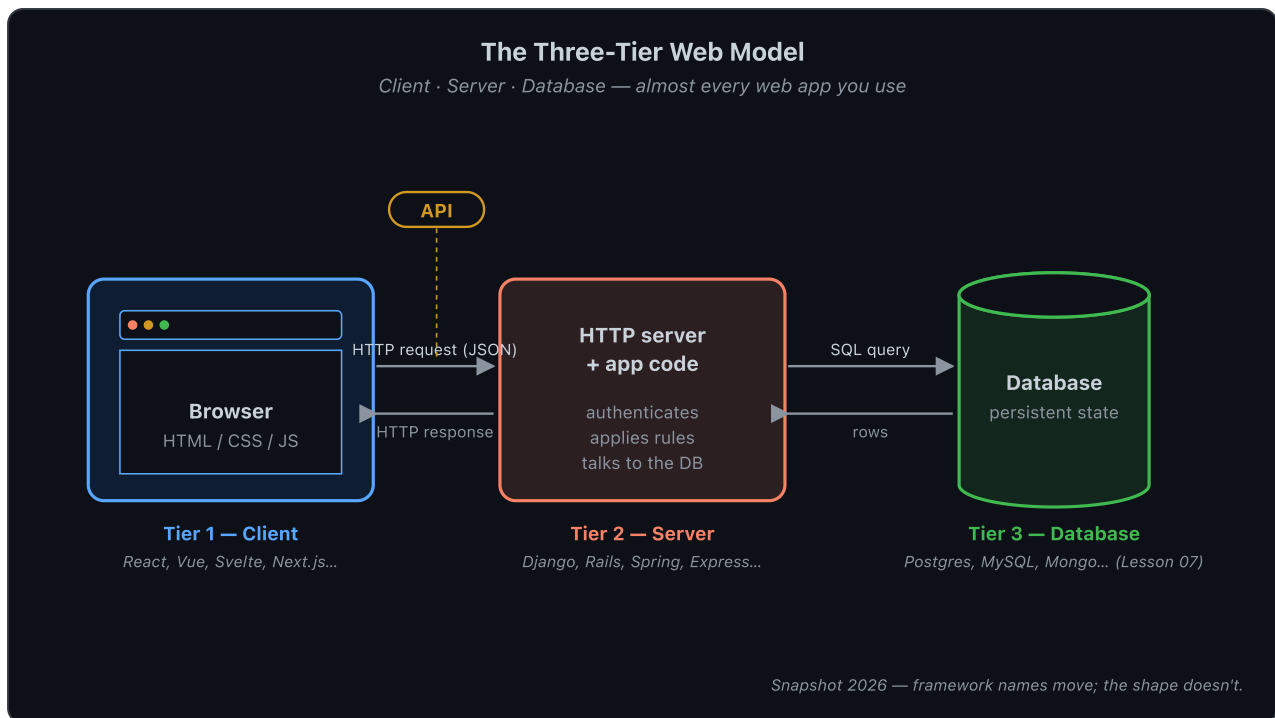
None of these is "the right one" — each language community has matured tooling and shipped real products at huge scale. Companies pick based on what their team already knows, what their hiring pool looks like, and what their product needs. The backend's job is the same regardless of language: receive a request, decide if the user is allowed to do this, talk to the database, return a response.

5. APIs — the contract between tiers

An **API** (Application Programming Interface) is a contract: *"ask me this way, I'll answer this way."* Most modern web APIs are **REST** — HTTP verbs (`GET`, `POST`, `PUT`, `DELETE`) plus URL paths plus JSON bodies. For example, `GET /api/users/42` returns `{ "id": 42, "name": "Ada" }`. A growing share use **GraphQL**, which exposes one endpoint and lets the client say what shape of data it wants. Newer **gRPC** is common for service-to-service traffic inside a company, where speed matters more than human readability. The big point: APIs aren't a separate technology — they're how the frontend and backend agree to talk, and how third-party apps (your phone's Slack app, Zapier, the AWS CLI) talk to a service from the outside.

6. The picture

Here is the whole three-tier model in one diagram. Notice the API label on the arrow between the client and the server — that's the contract you just read about.



7. Mobile, briefly

Mobile apps come in three flavors. **Native** apps are built per platform — **Swift** for iOS, **Kotlin** for Android — and feel most at home on the device, at the cost of maintaining two separate codebases. **Hybrid / cross-platform** uses one codebase to target both: **React Native** (Meta) and **Flutter** (Google) are the two large frameworks as of 2026. **Mobile web** is just a website your phone visits — sometimes installable to the home screen as a **PWA** (Progressive Web App), which can run offline and send notifications. All three flavors usually talk to the same backend and the same API as the web app, which is precisely the payoff of clean tiers in the first place.

What people argue about

Three live debates in modern web development right now:

- **REST vs GraphQL.** GraphQL solves real problems (overfetching, underfetching, one round trip for a complex screen) but adds server-side complexity and a learning curve; REST critics counter that GraphQL is "just

rebuilding SQL over HTTP, badly". Both still ship in production at scale and neither is winning outright.

- **Server-side rendering vs single-page app.** For a decade SPAs (single-page apps in React, Vue, Angular) replaced server-rendered HTML; now the pendulum is swinging back toward server rendering (**Next.js**, **Remix**, **Astro**, **Phoenix LiveView**) for faster first paint and better SEO. Expect the answer to keep shifting.
- **TypeScript everywhere.** The JavaScript world has largely adopted **TypeScript** (JavaScript with type annotations); Python and Ruby are slowly adding optional typing. Some argue the type-safety tax is too high for small projects and the tooling cost real; most teams who tried it didn't go back.

Reading on

- **MDN Web Docs** — developer.mozilla.org. The canonical free reference for HTML, CSS, and JavaScript, run by Mozilla. If a browser feature exists, MDN documents it.
- **web.dev** — web.dev. Google's home for modern web best-practices: performance, accessibility, PWAs, Core Web Vitals. Short, opinionated, current.
- **Lesson 06 of this course** — returns to the backend at a higher zoom level (monolith vs microservices vs serverless vs event-driven). Read this lesson first, then 06 will make sense.

LAST-REVIEWED: 2026-05-25