Practical Web Development

From Server-Rendered Pages to React with Python and FastAPI

Igor Benav

Contents

Preface

To be written.

About This Book

Most web development resources fall into two camps: tutorials that get you copying code without understanding why, or comprehensive references that bury you in details before you can build anything. This book tries to find the middle ground.

We start with how the web actually works (HTTP, requests, responses) before writing any server code. This isn't filler. When something breaks later, you'll know where to look because you understand the underlying mechanics.

The book is opinionated. We use Python and FastAPI on the server. For interactivity, we start with HTMX (server-rendered HTML, minimal JavaScript) before moving to React (client-rendered, more JavaScript). You'll understand both approaches and when each makes sense. We're not trying to cover every framework or teach you the "best" way. We're teaching you one coherent way that works, so you have solid ground to stand on when you explore other options later.

By the end, you'll have built and deployed a real web application. Not a toy project that only runs on your laptop, but something on the internet that other people can use.

How to Use This Book

This book assumes you already know Python. You should be comfortable with variables, functions, loops, conditionals, lists, and dictionaries. If you've worked through an introductory Python book or course, you're ready. We won't explain what a function is, but we will explain what a web server is.

The chapters are meant to be read in order. Each one builds on the previous. If you skip the chapter on HTTP, the chapter on forms won't make sense. If you skip templates, you'll be lost when we add HTMX.

Each chapter follows a pattern:

- **Concepts first**: What are we trying to do and why?
- **Implementation**: How to actually do it in code
- **Hands-on project**: A practical exercise that uses what you just learned
- **Exercises**: More practice on your own

Type the code yourself. Don't copy and paste. The errors you make and fix along the way are part of learning. Change things. Break things. See what happens.

When you get stuck, resist asking AI for the solution. Ask for a hint if you need to, but do the thinking yourself. Struggling is normal. It's also how you actually learn, rather than just feeling like you're learning.

Prerequisites

You'll need:

- Python 3.11 or newer installed
- A code editor (VS Code works well, I like Zed)
- A terminal you're comfortable using
- Basic Python knowledge (variables, functions, loops, lists, dictionaries)
- Willingness to read error messages instead of panicking

You don't need any prior web development experience. You don't need to know HTML, CSS, or JavaScript. We'll cover what you need as we go.

1 Chapter 1: How the Web Works

Web development is often described as "making websites," but that undersells what's happening. When you build for the web, you're writing software that runs across multiple computers: your server, the user's browser, maybe a database somewhere else. These machines communicate over a network, passing data back and forth. The "website" is just the visible result.

This distributed nature is what makes web development different from writing a script that runs on your laptop. Your code doesn't control everything. You write software that handles requests as they arrive and generates appropriate responses, trusting the network to deliver them.

Before you write any code for a website, you need to understand what happens when someone visits one. Not only because it's interesting background reading, but because you'll be confused later if you skip this. Let's start by understanding the main entities that are involved in this exchange: client and server.

1.1 Clients and Servers

The web runs on a simple relationship: one computer asks for something, another computer provides it.

The computer doing the asking is the client. Usually this is a web browser like Chrome, Firefox, Safari - running on someone's laptop or phone. The client requests resources (pages, images, data) and displays them to the user.

The computer providing resources is the server. It's just a computer running software that listens for requests and sends back responses. Could be a massive machine in a data center, could be your laptop. The hardware doesn't matter. What makes it a server is that it waits for requests and responds to them.

When you type `www.example.com` into your browser, your browser first figures out which server to contact, then sends a request to that server. The server processes the request, prepares a response, and sends it back. Your browser receives the response and renders it on screen. This happens every time you visit a page, click a link, or submit a form.

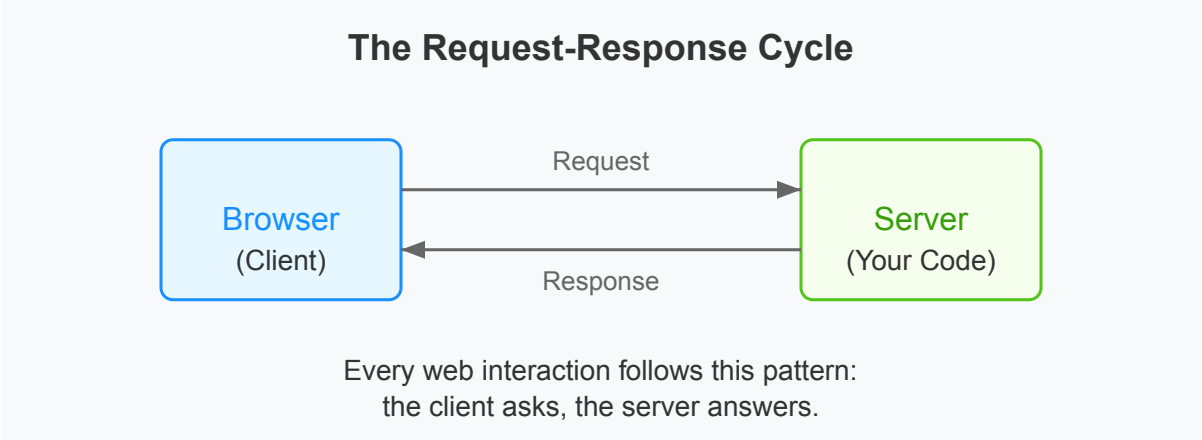

Figure 1.1: The Request-Response Cycle

But how does your browser know where to send the request? When you type `www.example.com`, your browser doesn't actually know where that is. It only understands numerical addresses called IP addresses, something like `93.184.216.34`.

The translation happens through DNS (Domain Name System), which works like a phone book for the internet. Your browser asks a DNS server "what's the IP address for www.example.com?" and gets back the number it needs.

You don't need to understand DNS deeply to build websites. But knowing it exists explains why sometimes a "website is down" when really the site is fine—nobody can find its address. Now let's understand better how clients and servers communicate.

1.2 HTTP: Requests and Responses

Once your browser knows where the server is, it needs a language to communicate. That language is HTTP: HyperText Transfer Protocol.

HTTP is simple. A request is text that says what you want. A response is text (plus possibly some data) that gives you what you asked for or explains why you can't have it.

When your browser requests a web page, it sends something like this:

```
GET /about HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) Firefox/120.0
Accept: text/html
```

`GET` is the HTTP method: what kind of action you want. GET means "give me this resource." POST means "here's some data to process." There are others (PUT, DELETE), but GET and POST handle most of what you'll do.

`/about` is the path: which resource you want on this server. The server uses this to decide what to send back.

`HTTP/1.1` is the protocol version. You may just ignore it for now.

The rest are headers: extra information about the request. `Host` says which website you want (one server can host multiple sites). `User-Agent` identifies your browser. `Accept` says what kind of content you can handle.

The server receives this request and sends back a response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 1256

<!DOCTYPE html>
<html>
<head>
    <title>About Us</title>
</head>
<body>
    <h1>About Our Company</h1>
    <p>We make things...</p>
</body>
</html>
```

`200 OK` is the status code, saying it worked. You've seen 404 (not found) in your life before. The ones you'll deal with most are these:

- 200 - OK, here's what you asked for
- 301/302 - This moved, go look over there (redirect)
- 400 - Your request didn't make sense
- 401 - You need to log in
- 403 - You're logged in but can't access this
- 404 - Doesn't exist
- 500 - Something broke on the server

The headers tell the browser what's coming. `Content-Type` says it's HTML. `Content-Length` says how many bytes. After the headers, a blank line, then the actual content.

When you build a web application, part of what you're writing is the server side of this conversation. Your code receives requests and decides what to send back. That's the job.

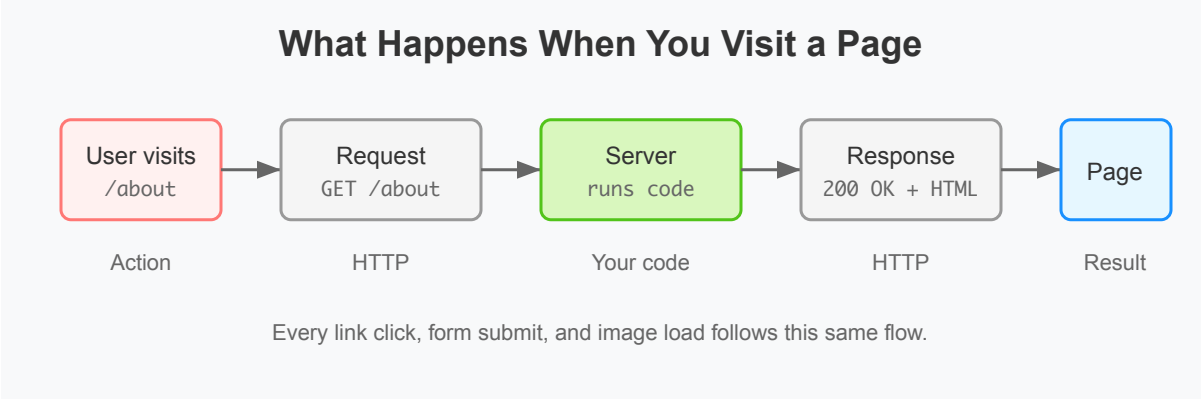

Figure 1.2: What Happens When You Visit a Page

Next, let's analyze the actual anatomy of URLs.

1.3 URLs

Every resource on the web has an address. Let's take one apart:

`https://www.example.com:443/products/shoes?color=red&size=10#reviews`

Looking at it in detail:

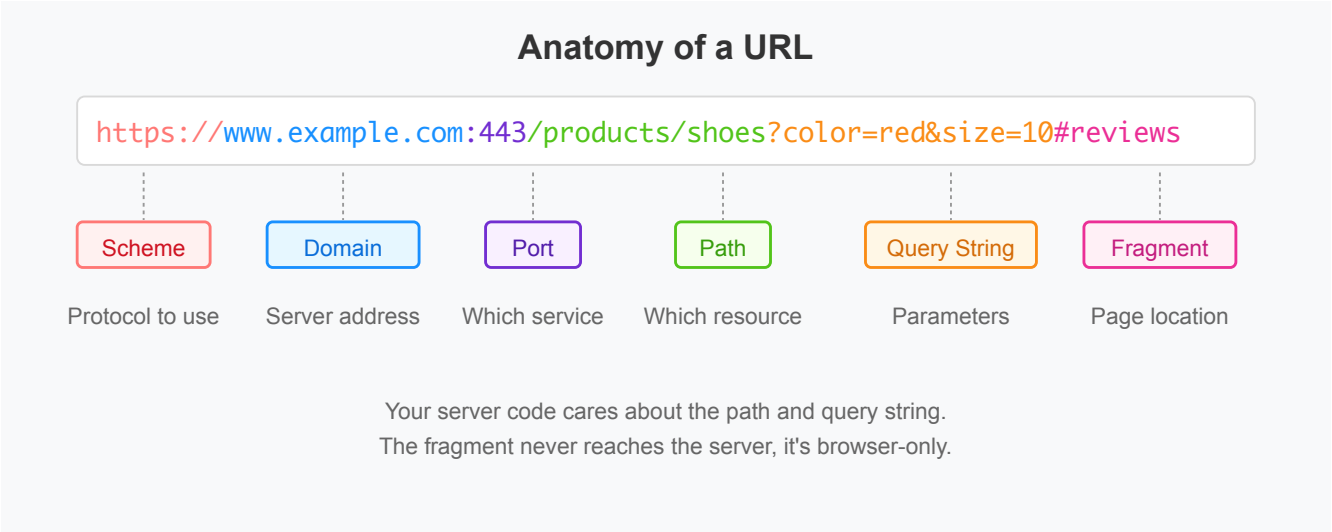

Figure 1.3: Anatomy of a URL

`https://` is the scheme. HTTPS is HTTP with encryption. You'll also see plain `http://` (unencrypted, increasingly rare).

`www.example.com` is the domain—the human-readable name that DNS translates to an IP address.

`:443` is the port. Think of the IP address as a building's street address and the port as the apartment number. One server can run multiple services on different ports. 443 is the default for HTTPS, so browsers hide it.

`/products/shoes` is the path. This tells the server which resource you want. When you build a web application, you write code that looks at this path and decides what to do. This is called routing.

`?color=red&size=10` is the query string. Parameters sent to the server. The `?` marks the start, `&` separates parameters, each parameter is `key=value`. Used for search terms, filters, pagination.

`#reviews` is the fragment. This never gets sent to the server. It tells the browser where to scroll on the page.

When you're building a server, you care about the path and query string. The path determines which code runs. The query string provides input to that code.

1.4 Static vs Dynamic Websites

Not all websites work the same way.

A static website is files on a server. When you request `/about.html`, the server finds that file and sends it. No code runs to generate the response—it just serves what's on disk. Static sites are simple, fast, and cheap to host. They work for content that doesn't change based on who's viewing it (blogs, documentation, portfolios).

But what if you want to show different content to different users? What if you need today's date, or the user's name, or results from a database? A static file can't do that.

A dynamic website runs code to generate each response. When you request `/profile`, the server doesn't look for a file called `profile`. Instead, it runs a program that checks if you're logged in, looks up your information in a database, generates HTML with your specific data, and sends that HTML back. The response might be different for every user, every time.

This is server-side programming: code that runs on the server, handles requests, and generates responses. You can do this in many languages (JavaScript, Ruby, Go, PHP). We're using Python with a framework called FastAPI.

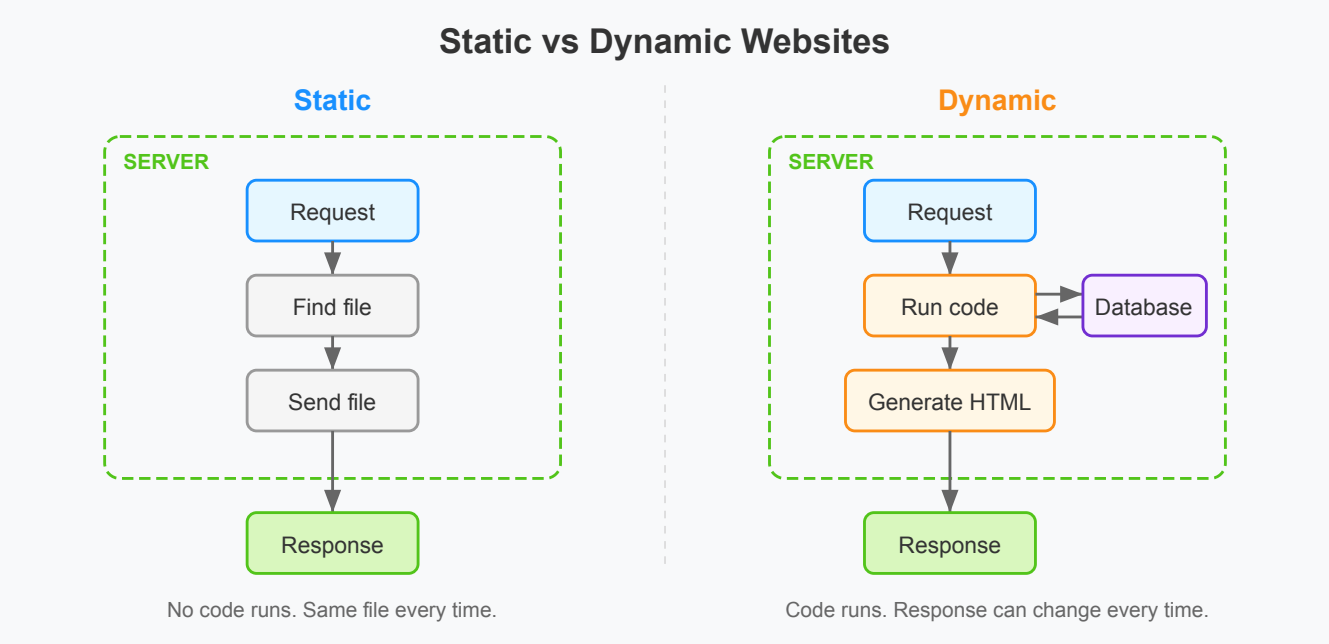

Figure 1.4: Static vs Dynamic Websites

Static sites work when content is the same for everyone, updates are infrequent, and you want maximum speed at minimum cost. Dynamic sites are necessary when content depends on who's viewing it, users can create or modify data, or you need real-time information.

Most applications are dynamic. That's what we're building. Next we'll see it in practice.

1.5 Looking at Network Traffic

Everything we've talked about isn't abstract - you can watch it happen. Every browser has developer tools that show the HTTP traffic between your browser and servers.

To open them:

- Chrome/Edge/Firefox: F12 or Ctrl+Shift+I (Cmd+Option+I on Mac)
- Safari: Enable in preferences first, then Cmd+Option+I

Go to the Network tab. Visit any website. Watch the requests appear.

You'll see the HTML document load first, then requests for CSS, JavaScript, images, fonts. Click any request to see the headers, status code, and response content.

This isn't just educational. When something isn't working, the Network tab tells you what's actually being sent and received. You'll use it constantly to debug.

1.6 What We're Building

Now that you understand the mechanics, here's where we're headed.

We start with HTML and CSS, the languages that define what pages contain and how they look. These are what your server sends to browsers.

Then we build servers with Python and FastAPI. You'll write code that receives HTTP requests and sends back responses. First simple HTML, then templates to generate complex pages from data.

We'll handle forms and user input: how data flows from browser to server.

For interactivity, we'll look at two approaches. HTMX lets you update parts of a page without writing JavaScript—the server still generates HTML, but the browser swaps it in without reloading. Then React, where the browser takes over more work and your server becomes an API that returns data instead of HTML. You'll understand both and know when to use which.

We'll add databases so applications can remember things, and authentication so users can have accounts.

Finally, we'll deploy to a real server so anyone can use what you've built.

1.7 Hands-On: Exploring HTTP with Browser Dev Tools

Let's put this into practice.

Exercise 1: Inspect a Page Load

Open developer tools (F12), go to the Network tab, and visit `http://example.com`. Find the first request—the HTML document. What's the HTTP method? Status code? Content-Type header? Response size?

Exercise 2: See a 404

Keep the Network tab open and visit `http://example.com/this-does-not-exist`. Check the status code. Look at the response body—servers usually send a custom "not found" page.

Exercise 3: Query Parameters

Go to any search engine and search for something. Look at the URL—find the query string. In the Network tab, see how the browser parsed the parameters.

Exercise 4: Watch Resources Load

Clear the Network tab and visit a news site or something with lots of content. Watch the requests pile up. Sort by type—count the HTML, CSS, JS, and image requests. A single "page" requires many HTTP requests. Each one is a complete request-response cycle.

1.8 Chapter Summary

- The web is clients (browsers) requesting resources from servers
- HTTP is the protocol—text requests and responses
- Requests have a method (GET, POST) and path; responses have a status code and content
- URLs contain everything needed to find a resource: scheme, domain, port, path, query string
- Static sites serve files as-is; dynamic sites run code to generate responses
- Browser developer tools let you see HTTP traffic

In the next chapter, we start working with HTML: the format most HTTP responses contain.

1.9 Exercises

1. Using developer tools, find three different HTTP status codes on real websites. Which sites, which codes, what caused them?
2. Look at the User-Agent header your browser sends. What does each part mean?
3. Pick a website you use often. How many HTTP requests does the homepage make? How many different domains do they go to?
4. Find a site with search. Do a search, look at the URL, find the query parameter with your search term. Modify the URL to do a different search.
5. Compare response headers from `example.com` (static) versus a news site (dynamic). What differences do you notice? Look for `Cache-Control`, `Set-Cookie`, `Server`.

References

Downey, Allen B. 2015. *Think Python: How to Think Like a Computer Scientist*. 2nd ed. O'Reilly Media.

Guttag, John. 2016. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. 2nd ed. MIT Press.

Hammack, Richard H. 2020. *Book of Proof*. 3rd ed. Virginia Commonwealth University.

Percival, Harry J. W., and Bob Gregory. 2020. *Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices*. O'Reilly Media.

Pólya, George. 2014. *How to Solve It: A New Aspect of Mathematical Method*. 2nd ed. Princeton University Press.

Ramalho, Luciano. 2021. *Fluent Python: Clear, Concise, and Effective Programming*. 2nd ed. O'Reilly Media.

Robson, Patrick. 2021. *Robust Python: Write Clean and Maintainable Code*. O'Reilly Media.

Shaw, Zed A. 2013. *Learn Python the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code*. 3rd ed. Zed Shaw's Hard Way Series. Addison-Wesley Professional.

Slatkin, Brett. 2019. *Effective Python: 125 Specific Ways to Write Better Python*. 2nd ed. Effective Software Development Series. Addison-Wesley Professional.

Sweigart, Al. 2019. *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. 2nd ed. No Starch Press.

Velleman, Daniel J. 2019. *How to Prove It: A Structured Approach*. 3rd ed. Cambridge University Press.