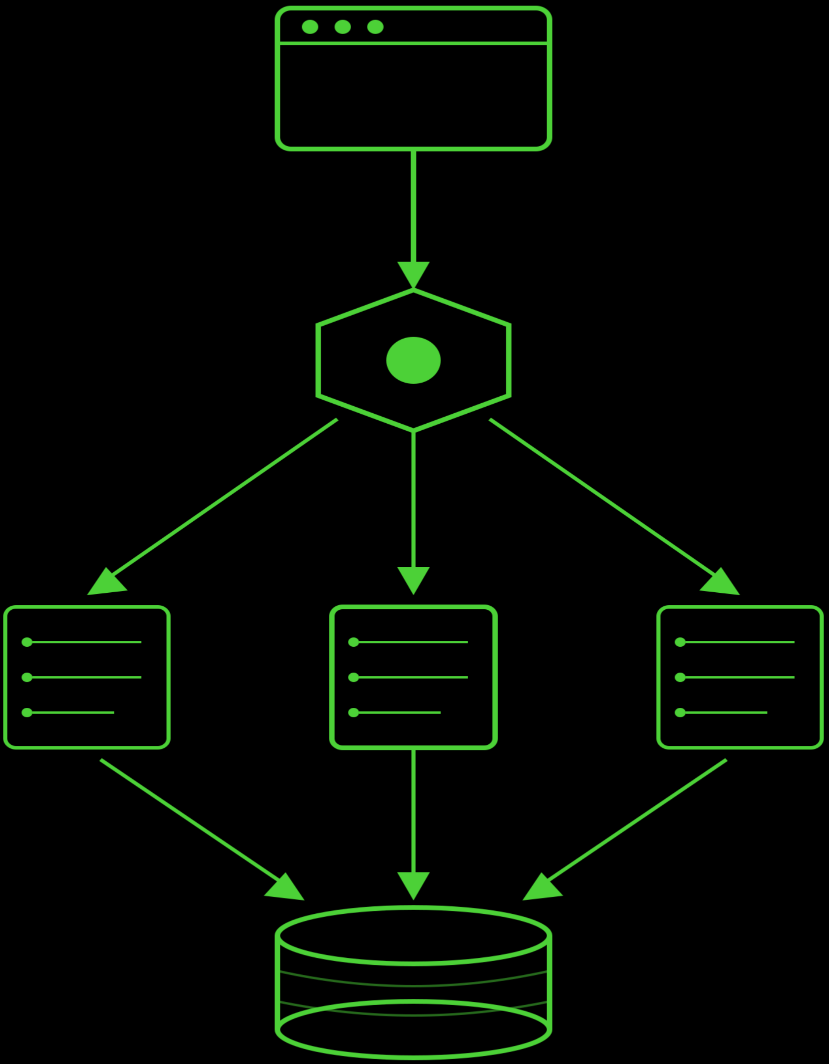
Practical Web Development

From Server-Rendered Pages to React with Python and FastAPI



Igor Benav

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). 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.

What You'll Build

The book follows a single thread: 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. Here's the path we take to get there.

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, so you need to understand them before writing any server code.

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. Then for interactivity, we'll look at two approaches. HTMX lets you update parts of a page without writing JavaScript. React moves more work to the browser and turns your server into 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, authentication so users can have accounts, and finally deploy to a real server so anyone can use what you've built.

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.

The complete code for each chapter's hands-on project is available at github.com/Applied-Computing-League/practical-web-development. Consider it a last resort. If you look at the solution before genuinely struggling with the problem, you're only cheating yourself out of the 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.

Contributors

Name	Role
Igor Benav	Author

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.

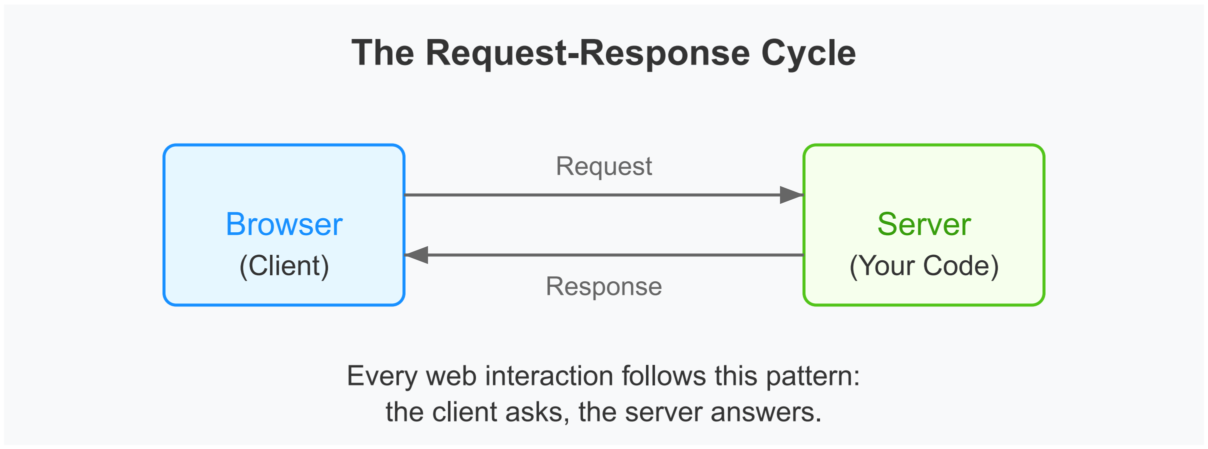

The Request-Response Cycle
Browser
(Client)
Request
Response
Server
(Your Code)
Every web interaction follows this pattern:
the client asks, the server answers.

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. Once the browser knows where the server is, though, it needs a way to talk to it.

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.

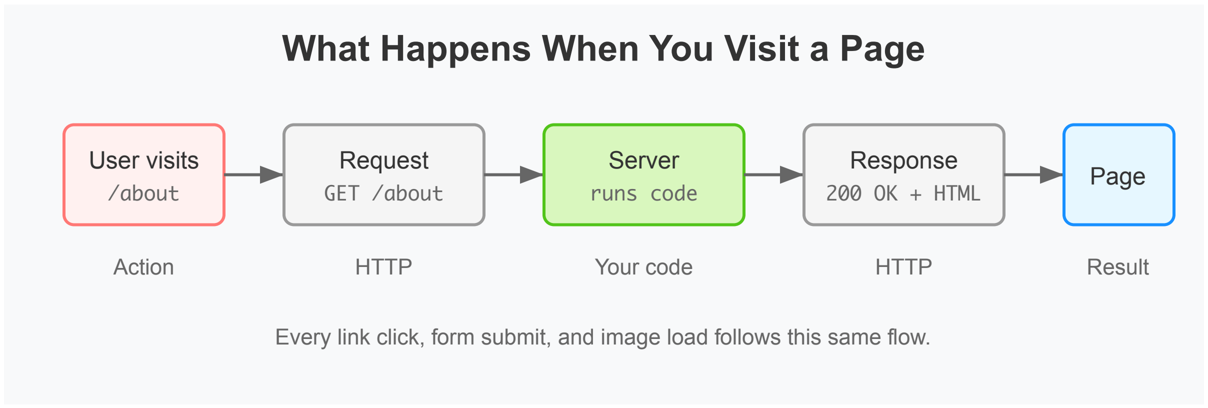


Figure 1.2: What Happens When You Visit a Page

Every request targets a specific URL. We've been looking at simple ones like /about, but URLs can carry a lot more information.

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:

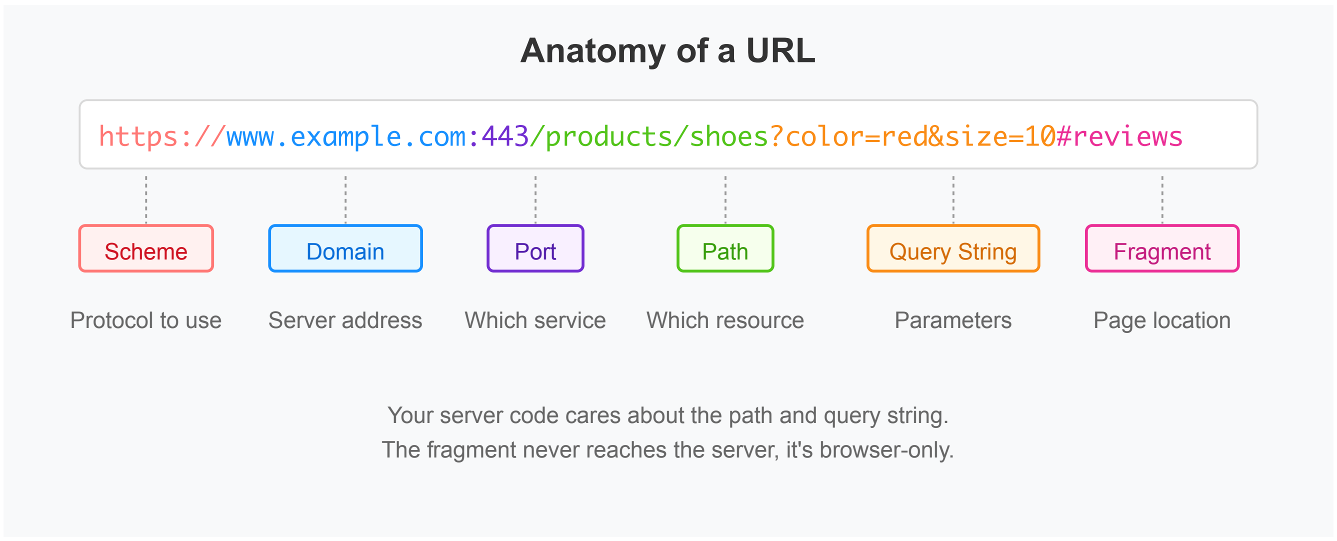

Anatomy of a URL
https://www.example.com:443/products/shoes?color=red&size=10#reviews
Scheme
Domain
Port
Path
Query String
Fragment
Protocol to use
Server address
Which service
Which resource
Parameters
Page location
Your server code cares about the path and query string.
The fragment never reaches the server, it's browser-only.

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.

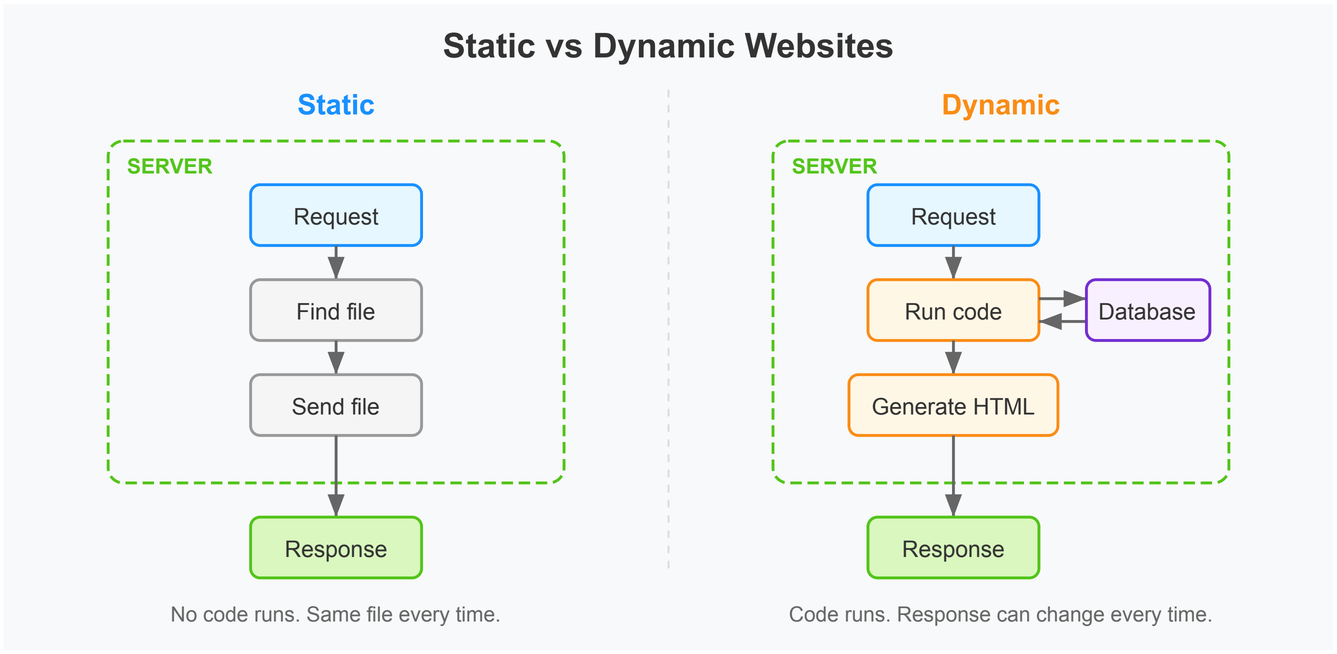

Static vs Dynamic Websites
Static
SERVER
Request
Find file
Send file
Response
No code runs. Same file every time.
Dynamic
SERVER
Request
Run code
Database
Generate HTML
Response
Code runs. Response can change every time.

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.

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 Hands-On: Exploring HTTP with Browser Dev Tools

Open your browser and follow along.

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.

1.7 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.

1.8 Exercises

1. Using developer tools, find three different HTTP status codes on real websites. Which sites, which codes, what caused them?
2. Open two different browsers (or a browser and a private/incognito window) and visit the same page. Compare the request headers in the Network tab. What's different between them? What's the same? Pay attention to Accept, Accept-Language, and User-Agent.

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. Visit example.com and a site you're logged into (email, social media, anything). Compare the response headers. The static site's headers are simple. The dynamic site's are more complex. What extra headers does the dynamic site send, and why might it need them? (Hint: think about what a site that knows who you are needs to do differently.)

2 Chapter 2: HTML

In the previous chapter, we saw what the server sends back when you visit a webpage: text that starts with `<!DOCTYPE html>` and contains things like `<head>`, `<body>`, and `<p>`. That text is HTML, and it's what this chapter is about.

HTML is not a programming language. You can't write loops or conditionals in it. It's a markup language: a way to describe the structure of a document. When you write HTML, you're telling the browser "this is a heading," "this is a paragraph," "this is a link to another page." The browser reads those instructions and renders them visually.

Every webpage you've ever visited is built on HTML. The browser might also load CSS (for styling) and JavaScript (for interactivity), but the foundation is always HTML. If you right-click on any webpage and select "View Page Source," you'll see the HTML that built it. But what even is HTML?

2.1 What HTML Is

HTML stands for HyperText Markup Language. The "HyperText" part refers to links: text that connects to other documents. The "Markup" part refers to the tags you use to annotate content.

An HTML document is made of elements. Each element usually has an opening tag, some content, and a closing tag:

```
<p>This is a paragraph.</p>
```

`<p>` is the opening tag. `</p>` is the closing tag (note the forward slash, `/`). Everything between them is the content of the element.

Some elements don't have content and don't need a closing tag:

```
<br>
<img src="photo.jpg">
```

`<br>` is a line break. `<img>` displays an image. These are called "void elements" or "self-closing elements."

Elements can have attributes, which provide extra information:

```
<a href="https://example.com">Click here</a>
<img src="photo.jpg" alt="An Ipanema sunset">
```

The `href` attribute tells the link where to go. The `src` attribute tells the image which file to load. The `alt` attribute provides text for screen readers and for when the image fails to load. Attributes always go in the opening tag, and they follow the pattern `name="value"`.

So far these are fragments. A real HTML page needs a bit more structure around them.

2.2 A Complete HTML Document

Here's the minimal structure of an HTML page:

```
<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
</head>
<body>
    <h1>Hello, world!</h1>
    <p>This is my first webpage.</p>
</body>
</html>
```

`<!DOCTYPE html>` tells the browser this is an HTML5 document. It's not technically a tag, just a declaration that should be at the top of every HTML file.

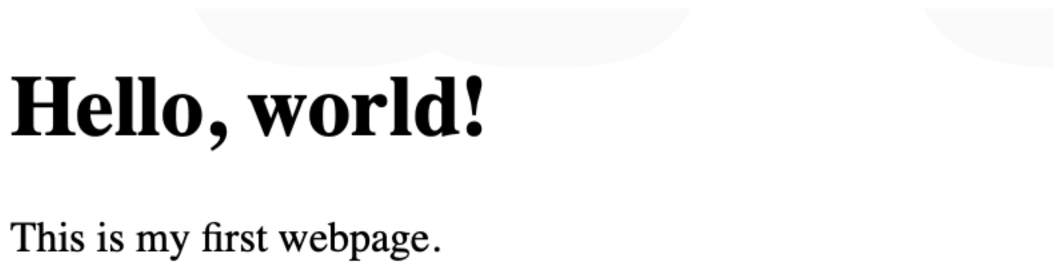
`<html>` is the root element. Everything else goes inside it.

`<head>` contains metadata about the page: the title (which appears in the browser tab), links to stylesheets, and other information that doesn't appear on the page itself.

`<body>` contains everything visible on the page. This is where your actual content goes.

`<title>` sets what appears in the browser tab and in search results. It's important but easy to forget.

To see this in action, create a file called `index.html` on your computer, paste the code above into it, and open it in your browser. You should see "Hello, world!" as a heading and "This is my first webpage." as a paragraph. Here's what you should see:


Hello, world!

This is my first webpage.

Figure 2.1: A basic HTML page rendered in the browser

2.3 Essential Tags

You don't need to memorize every HTML tag. There are over a hundred, and you'll use maybe twenty regularly (plus you'll usually ask AI for some of it). It's useful knowing these ones to understand the structure better:

Headings go from `<h1>` (most important) to `<h6>` (least important):

```
<h1>Main Title</h1>
<h2>Section Title</h2>
<h3>Subsection Title</h3>
```

Use `<h1>` once per page for the main title. Use `<h2>` for major sections, `<h3>` for subsections within those. Don't skip levels just because you want smaller text (that's what CSS is for).

Paragraphs are wrapped in `<p>` tags:

```
<p>This is the first paragraph.</p>
<p>This is the second paragraph.</p>
```

Without `<p>` tags, the browser ignores line breaks in your HTML and runs everything together.

Links use the `<a>` tag (the "a" stands for anchor):

```html
<a href="https://example.com">Visit Example</a>
<a href="/about">About Us</a>
<a href="#section2">Jump to Section 2</a>
```

The `href` attribute is the destination. It can be a full URL, a path on the same site, or a fragment identifier (starting with `#`) to jump to an element on the current page.

Images use the `<img>` tag:

```html
<img src="photo.jpg" alt="Description of the image">
```

The `src` attribute is the path to the image file. The `alt` attribute is required for accessibility (screen readers read it aloud, and it displays if the image fails to load), plus it helps a lot with SEO - Search Engine Optimization (making your website easier to find in search engines like google).

Lists come in two flavors. Unordered lists (bullet points):

```html
<ul>
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
</ul>
```

And ordered lists (numbered):

```html
<ol>
    <li>First step</li>
    <li>Second step</li>
    <li>Third step</li>
</ol>
```

`<ul>` stands for unordered list, `<ol>` for ordered list, and `<li>` for list item.

Divisions and spans are generic containers:

```html
<div>
    <p>This paragraph is inside a div.</p>
    <p>So is this one.</p>
</div>

<p>This word is <span>highlighted</span> somehow.</p>
```

`<div>` is a block-level container (takes up the full width). `<span>` is an inline container (flows with the text). On their own, they don't do anything visible. They're useful for grouping elements so you can style them with CSS or manipulate them with JavaScript.

Line breaks and horizontal rules:

```html
<p>First line<br>Second line</p>
<hr>
<p>Content after the horizontal line</p>
```

`<br>` forces a line break within a paragraph. `<hr>` draws a horizontal line (historically "horizontal rule") to separate content.

2.4 Semantic Markup

You could build an entire website using only `<div>` and `<span>` tags. Wrap everything in divs, give them class names, and style them with CSS. It would look identical in a browser. A lot of early websites worked this way.

The problem is that a `<div class="header">` means nothing to anyone except you. Your browser doesn't know it's a header. A screen reader (software that reads pages aloud for visually impaired users) doesn't know it's a header. Google's search crawler doesn't know it's a header. They all see a generic box.

Semantic tags fix this by describing what content *means*, not just how it looks:

```html
<header>
    <h1>My Website</h1>
    <nav>
        <a href="/">Home</a>
        <a href="/about">About</a>
    </nav>
</header>

<main>
    <article>
        <h2>Article Title</h2>
        <p>Article content...</p>
    </article>
</main>

<footer>
    <p>&copy; 2025 My Website</p>
</footer>
```

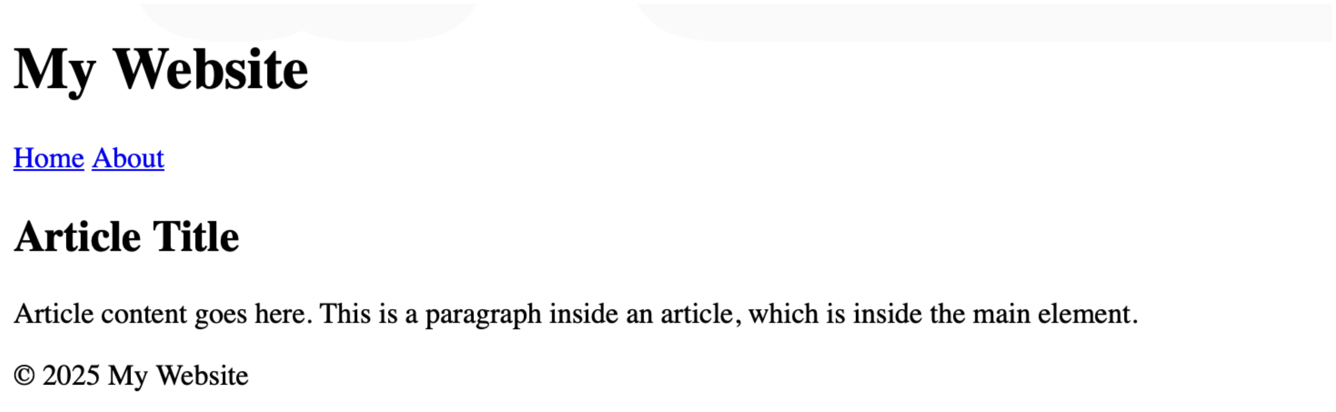
`<header>` is for introductory content, usually at the top. `<nav>` is for navigation links. `<main>` is for the primary content of the page. `<article>` is for self-contained content that could stand alone. `<footer>` is for content at the bottom (copyright, contact info).

Compare that to the div-only version:

```html
<div class="header">
    <h1>My Website</h1>
    <div class="nav">
        <a href="/">Home</a>
```

```
            <a href="/about">About</a>
        </div>
    </div>
```

In the browser, it looks like this:



My Website

Home About

Article Title

Article content goes here. This is a paragraph inside an article, which is inside the main element.

© 2025 My Website

Figure 2.2: Semantic HTML elements rendered in the browser

Both render the same way. The difference is in what happens behind the visible page. A screen reader can skip straight to `<main>` so a blind user doesn't have to listen to the navigation on every page. Google can identify the `<article>` as the important content and use it in search results. Your future self (or a teammate) can read the HTML and immediately understand the page structure without deciphering class names.

This isn't theoretical. Search engine rankings are affected by how well your HTML communicates structure. Accessibility lawsuits have been filed against companies whose websites are unusable with screen readers. Using semantic tags costs you nothing and helps on both fronts.

A few more semantic tags worth knowing:

```
<section>  <!-- A thematic grouping of content -->
<aside>    <!-- Content tangentially related to the main content -->
<figure>   <!-- Self-contained content like images with captions -->
<figcaption>  <!-- Caption for a figure -->
<time>     <!-- A date or time -->
<mark>     <!-- Highlighted text -->
<strong>   <!-- Important text (renders bold by default) -->
<em>       <!-- Emphasized text (renders italic by default) -->
```

Use `<strong>` when text is important, not just when you want bold. Use `<em>` when text is emphasized, not just when you want italics. The visual result is the same, but screen readers change their tone of voice for `<strong>` and `<em>`, which conveys meaning that bold and italic styling alone can't.

2.5 Links and Navigation

Links are what make the web a web.

A basic link:

```html
<a href="https://example.com">External Site</a>
```

Linking to a page on the same site:

```html
<a href="/about">About Us</a>
<a href="/contact">Contact</a>
```

The leading / means "start from the root of the site." So if your site is at `example.com`, the link goes to `example.com/about`.

Linking to a section on the same page:

```html
<a href="#pricing">Jump to Pricing</a>

<!-- later in the document -->
<h2 id="pricing">Pricing</h2>
```

The `#pricing` in the href matches the `id="pricing"` on the target element. Clicking the link scrolls the page to that element.

Opening a link in a new tab:

```html
<a href="https://example.com" target="_blank">Opens in new tab</a>
```

Users generally expect to control whether links open in new tabs, so only use this when necessary.

Linking an image:

```html
<a href="/products">
    <img src="product.jpg" alt="Our product">
</a>
```

Any element can go inside an `<a>` tag, making the entire thing clickable.

A navigation menu is typically a list of links inside a `<nav>` element:

```html
<nav>
    <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/products">Products</a></li>
        <li><a href="/about">About</a></li>
        <li><a href="/contact">Contact</a></li>
    </ul>
</nav>
```

By default this renders as a bulleted list. With CSS you can remove the bullets and arrange the links horizontally or vertically as needed.

2.6 Forms

Forms are how users send data back to the server. When you log in, search for something, or submit a comment, you're using a form.

A simple form:

```
<form action="/search" method="GET">
    <input type="text" name="q" placeholder="Search...">
    <button type="submit">Search</button>
</form>
```

The `action` attribute is where the form data gets sent (a URL). The `method` is the HTTP method to use (GET or POST, which we covered in Chapter 1).

When you submit this form, the browser makes a request to `/search?q=whatever` where "whatever" is what you typed in the input field.

The `name` attribute is critical. It becomes the key in the query string or form data. Without a name, the input's value won't be sent.

Text inputs:

```
<input type="text" name="username" placeholder="Username">
<input type="email" name="email" placeholder="Email address">
<input type="password" name="password" placeholder="Password">
<input type="number" name="age" placeholder="Age">
```

The `type` attribute changes the behavior. `email` shows an email keyboard on mobile and validates the format. `password` hides the characters. `number` only allows digits.

Labels make forms accessible:

```
<label for="username">Username</label>
<input type="text" id="username" name="username">
```

The `for` attribute matches the `id` of the input. Clicking the label focuses the input. Screen readers read the label when the input is focused.

You can also wrap the input inside the label:

```
<label>
    Username
    <input type="text" name="username">
</label>
```

This achieves the same effect without needing `for` and `id`.

Textareas are for multi-line text:

```
<label for="message">Message</label>
<textarea id="message" name="message" rows="5"></textarea>
```

Select dropdowns:

```
<label for="country">Country</label>
<select id="country" name="country">
    <option value="">Choose...</option>
    <option value="us">United States</option>
    <option value="uk">United Kingdom</option>
    <option value="ca">Canada</option>
</select>
```

The `value` attribute is what gets sent to the server. The text between the tags is what the user sees.

Checkboxes and radio buttons:

```
<label>
    <input type="checkbox" name="newsletter" value="yes">
    Subscribe to newsletter
</label>

<fieldset>
    <legend>Preferred contact method</legend>
    <label>
        <input type="radio" name="contact" value="email"> Email
    </label>
    <label>
        <input type="radio" name="contact" value="phone"> Phone
    </label>
</fieldset>
```

Radio buttons with the same `name` attribute form a group. Only one can be selected. Checkboxes are independent.

Submit buttons:

```
<button type="submit">Send</button>
<!-- or -->
<input type="submit" value="Send">
```

Both work. The `<button>` tag is more flexible because you can put other elements inside it.

A complete form might look like this:

```
<form action="/contact" method="POST">
    <div>
        <label for="name">Name</label>
        <input type="text" id="name" name="name" required>
    </div>

    <div>
        <label for="email">Email</label>
        <input type="email" id="email" name="email" required>
    </div>

    <div>
        <label for="message">Message</label>
        <textarea id="message" name="message" rows="5" required></textarea>
    </div>

    <button type="submit">Send Message</button>
</form>
```

The `required` attribute prevents submission if the field is empty. The browser handles this validation automatically. Here's how the form renders:

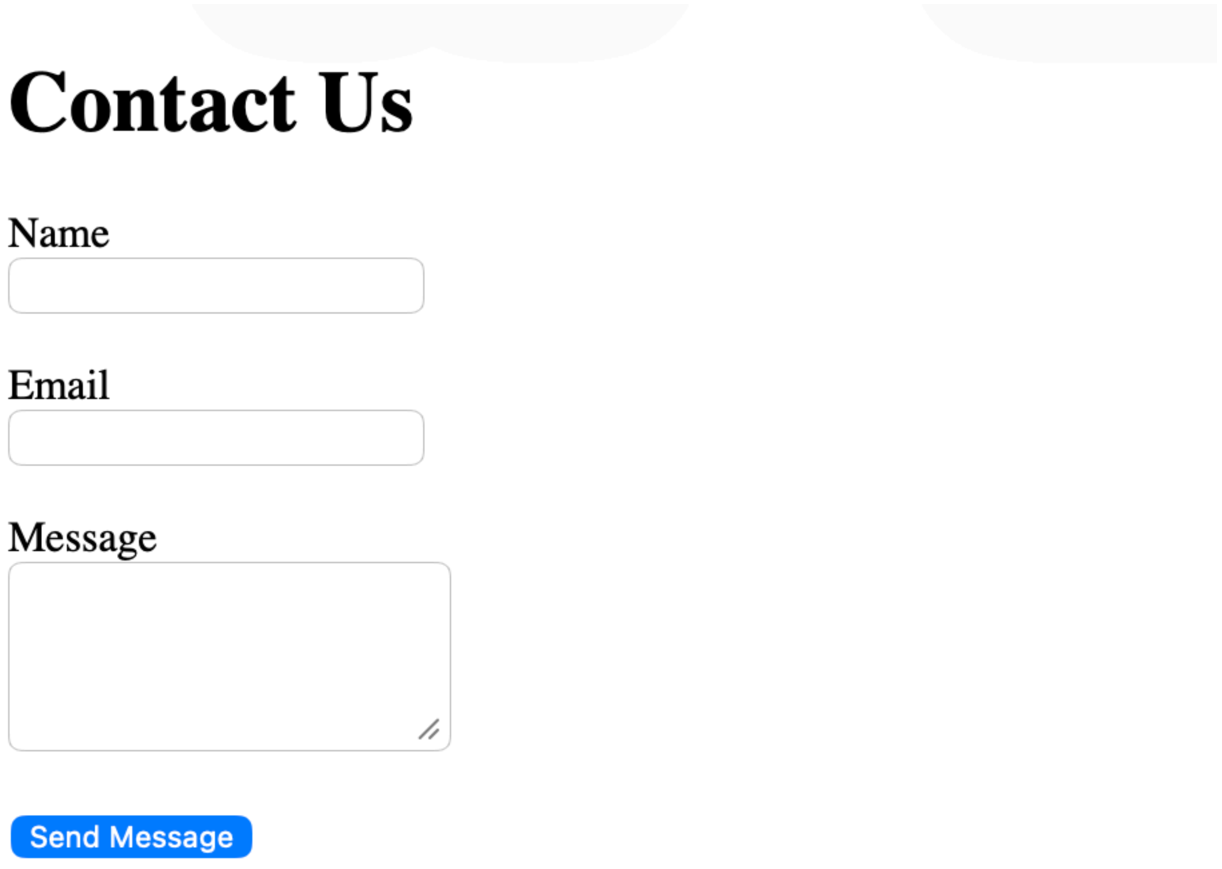


Figure 2.3: A contact form rendered in the browser

Plain, but functional. The styling comes later.

Forms are essential for web applications. We'll revisit them in Chapter 6 when we learn how to receive and process form data on the server.

2.7 Hands-On: Personal Profile Page

Let's build a complete HTML page that uses everything we've covered. Create a file called `profile.html` and build a personal profile page.

Your page should include:

1. A proper document structure (`<!DOCTYPE html>`, `<html>`, `<head>`, `<body>`)
2. A title that appears in the browser tab
3. A header with your name as an `<h1>` and a navigation menu
4. A main section with:
    - An "About Me" section with a few paragraphs
    - A "Skills" section with an unordered list
    - A "Contact" section with a simple contact form
5. A footer with copyright text

Here's a starting point:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Your Name - Profile</title>
</head>
<body>
    <header>
        <h1>Your Name</h1>
        <nav>
            <a href="#about">About</a>
            <a href="#skills">Skills</a>
            <a href="#contact">Contact</a>
        </nav>
    </header>

    <main>
        <section id="about">
            <h2>About Me</h2>
            <!-- Add paragraphs about yourself -->
        </section>

        <section id="skills">
            <h2>Skills</h2>
            <!-- Add an unordered list of skills -->
        </section>
        <section id="contact">
            <h2>Contact</h2>
            <!-- Add a contact form -->
        </section>
    </main>

    <footer>
        <!-- Add copyright -->
    </footer>
</body>
</html>
```

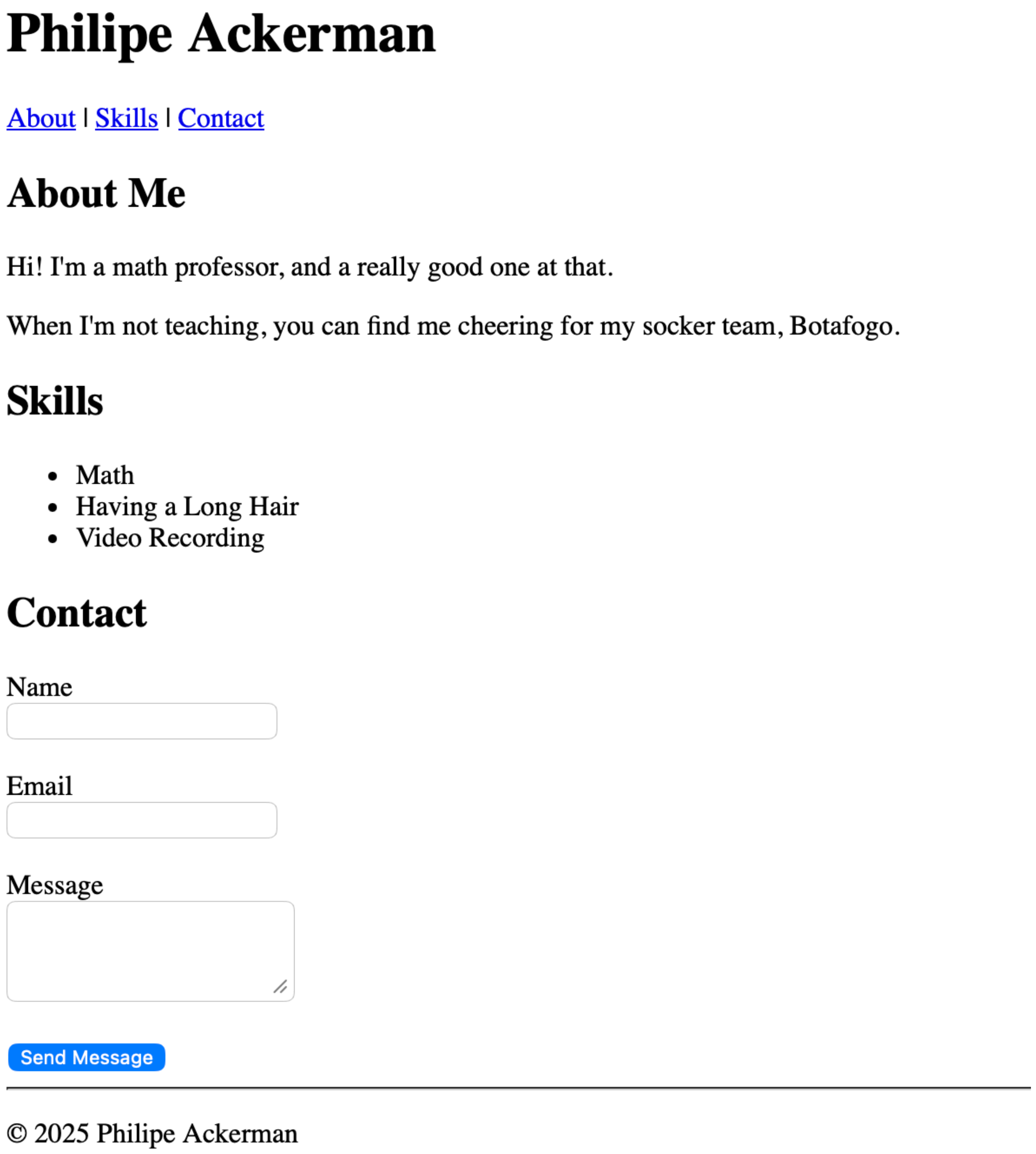
Fill in the blanks. Add real content about yourself (or about a character, be creative). The navigation links use fragment identifiers to jump to each section.

Stuck? Solution available

If you've given it an honest try and are truly stuck, a completed version of this project is available at github.com/Applied-Computing-League/practical-web-development/en/code/chapter2. Try to solve it yourself first. You learn more from the struggle than from reading the answer.

Open the file in your browser to see the result. It won't look pretty (that's what CSS is for), but the structure should be clear. Try viewing the page source in your browser. What you see should match what you wrote.

Click the navigation links and watch the page scroll. With the template above (and some content filled in), you'll see something like this:



Philipe Ackerman

About | Skills | Contact

About Me

Hi! I'm a math professor, and a really good one at that.

When I'm not teaching, you can find me cheering for my socker team, Botafogo.

Skills

- Math
- Having a Long Hair
- Video Recording

Contact

Name

Email

Message

Send Message

© 2025 Philipe Ackerman

Figure 2.4: The profile page template rendered in the browser

Ugly, I know. But, again, structure first, style later.

2.8 Chapter Summary

- HTML describes the structure of a document using tags
- Elements have opening tags, content, and closing tags: `<p>content</p>`
- Attributes provide extra information: `<a href="url">link</a>`
- Every page needs `<!DOCTYPE html>`, `<html>`, `<head>`, and `<body>`
- Semantic tags (`<header>`, `<main>`, `<article>`, `<nav>`) describe meaning
- Links (`<a>`) connect pages; forms (`<form>`) collect user input
- The `name` attribute on form inputs determines what gets sent to the server

The page you built works, but it's visually plain. In the next chapter, we'll add CSS to control colors, spacing, fonts, and layout.

2.9 Exercises

1. Add an image to your profile page. Find a photo online or use a placeholder service like `https://placekitten.com/200/200`. Remember the `alt` attribute.
2. Create a second HTML file called `projects.html` with a list of projects (real or imaginary). Add a link to it from your profile page's navigation, and add a link back to the profile page.
3. Expand your contact form: add a dropdown for "Reason for contact" with a few options, and add a checkbox for "Subscribe to newsletter." Then open the Network tab, fill out the form, and click submit. Look at the request that gets sent. Find your form data in the request. What happens to the checkbox value when it's checked vs unchecked? What about the dropdown?
4. Using only HTML (no CSS), try to create a simple table showing your weekly schedule. Look up the `<table>`, `<tr>`, `<th>`, and `<td>` tags.
5. View the source of example.com and one simple, well-structured site (a blog or documentation site works well). Find the semantic elements:

   ,

   ,

   ,

   . Then view the source of a complex site like a social media homepage. What's different? Why do you think large applications end up with so many nested

   elements?

6. Use the browser's developer tools (right-click, "Inspect") on your profile page. Try editing the HTML directly in the inspector. What happens when you change an element's text or delete a tag? (Don't worry, refreshing the page restores everything. You're changing stuff at the client.)

3 Chapter 3: CSS Basics

The profile page we built in the last chapter works, but it looks like it's from the 90s (you're probably not even old enough to remember). The headings are black, the background is white, the links are blue and underlined, and everything is aligned and cramped to the left. HTML gives you structure, but it doesn't give you control over how things look. That's what CSS is for.

CSS stands for Cascading Style Sheets. It's the language that controls colors, fonts, spacing, and layout. When you visit a website that looks good (or bad), that's CSS at work. HTML says "this is a heading." CSS says "this heading should be dark blue, 32 pixels tall, and have some space below it."

Like HTML, CSS isn't a programming language. You write rules that describe how elements should look, and the browser applies them. There are a few ways to attach those rules to your HTML.

3.1 How CSS Connects to HTML

There are three ways to add CSS to an HTML page.

Inline styles go directly on an element using the `style` attribute:

```
<p style="color: red; font-size: 18px;">This paragraph is red and larger.</p>
```

This works, but it's messy. If you want all your paragraphs to look the same, you'd have to copy that style attribute everywhere. And if you want to change it later, you'd have to find every instance to change.

Internal stylesheets go in a `<style>` tag in the `<head>`:

```
<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
    <style>
        p {
            color: red;
            font-size: 18px;
        }
    </style>
</head>
<body>
    <p>This paragraph is red and larger.</p>
    <p>So is this one.</p>
</body>
```

```
</html>
```

Now every paragraph on the page gets the same style. Change it once, it changes everywhere. This is already better, but the CSS is still tied to this specific HTML file.

External stylesheets put the CSS in a separate file, so we'd have a `style.css` file with:

```
/* styles.css */
p {
    color: red;
    font-size: 18px;
}
```

And a `index.html` file with:

```
<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <p>This paragraph is red and larger.</p>
</body>
</html>
```

The `<link>` tag tells the browser to load `styles.css` and apply it to this page. This is the approach we actually use for real projects. It keeps your HTML clean and lets multiple pages share the same styles.

For the rest of this chapter, I'll show CSS rules and not necessarily show the related HTML. Assume they're either in a `<style>` tag or an external file.

3.2 Selectors and Properties

A CSS rule has two parts: a selector (what to style) and a declaration block (how to style it):

```
h1 {
    color: navy;
    font-size: 32px;
}
```

`h1` is the selector. It matches all `<h1>` elements on the page.

Inside the curly braces are declarations. Each declaration has a property (`color`, `font-size`) and a value (`navy`, `32px`). Properties and values are separated by a colon (`:`). Declarations end with a semicolon (`;`).

Element selectors match HTML tags:

```
p {
    color: #333;
}

a {
    color: blue;
}

h1, h2, h3 {
    font-family: Georgia, serif;
}
```

That last one is a group selector. It applies the same style to `h1`, `h2`, and `h3` elements.

Class selectors match elements with a specific class attribute. They start with a dot:

```
<p class="intro">This is the introduction.</p>
<p>This is a regular paragraph.</p>
```

And creating the style:

```
.intro {
    font-size: 20px;
    font-weight: bold;
}
```

Only the paragraph with `class="intro"` gets the larger, bold text. Classes are reusable. You can put the same class on as many elements as you want.

ID selectors match a single element with a specific id. They start with a hash (#):

```
<header id="main-header">
    <h1>My Website</h1>
</header>
```

And creating the style that applies to the id:

```
#main-header {
    background-color: #f5f5f5;
    padding: 20px;
}
```

IDs should be unique on a page. Don't overuse this pattern, classes are more flexible and usually preferred.

Descendant selectors match elements inside other elements:

```
nav a {
    color: white;
    text-decoration: none;
}
```

This matches `<a>` elements that are inside a `<nav>`. Links elsewhere on the page aren't affected.

Combining selectors:

```
header nav a {
    color: white;
}

p.intro {
    font-size: 20px;
}
```

The first matches links inside nav inside header. The second matches paragraphs with the class "intro" (not all elements with that class, just paragraphs).

There are more selectors (attribute selectors, pseudo-classes, pseudo-elements), but these cover most of what you'll need.

3.3 Common Properties

CSS has hundreds of properties. You'll use maybe thirty regularly, and look up the rest when you need them. Rather than listing them all, let's style a real element and introduce properties as we go.

Start with a page that has some text and a card:

```
<body>
    <h1>My Website</h1>
    <p>Welcome to the site.</p>

    <div class="card">
        <h2>Latest Post</h2>
        <p>This is a preview of the post content...</p>
        <a href="/post/1">Read more</a>
    </div>
</body>
```

Without any CSS, this is black text on a white background in the browser's default font (usually Times New Roman). Everything is cramped against the left edge. Let's fix it piece by piece.

First, the typography. Set a font on the `body` and everything inside inherits it:

```
body {
    font-family: Arial, Helvetica, sans-serif;
    font-size: 16px;
    line-height: 1.5;
    color: #333;
}
```

`font-family` takes a list of fonts. The browser uses the first one it has, so you provide fallbacks. Always end with a generic family (`serif`, `sans-serif`, `monospace`) as the last resort.

`line-height` controls spacing between lines. A unitless number like `1.5` means 1.5 times the font size. This makes text much easier to read than the default (which is too tight).

The `color` property sets text color. `#333` is a dark gray hex code, softer than pure black. You can also use named colors (`red`, `navy`) or RGB values (`rgb(255, 0, 0)`), but hex codes are what you'll see most in real projects.

Now let's make the heading stand out:

```css
h1 {
    font-size: 32px;
    color: navy;
}
```

And style the card so it looks like an actual card:

```css
.card {
    background-color: #f9f9f9;
    border: 1px solid #ddd;
    border-radius: 8px;
    padding: 20px;
    max-width: 400px;
}
```

`background-color` does what you'd expect. `border` is shorthand for width, style, and color in one line. `border-radius` rounds the corners (higher values mean rounder). `padding` adds space between the card's edge and its content. `max-width` prevents the card from stretching across the entire page.

The link inside the card is the default blue with an underline. Let's change that:

```css
.card a {
    color: #2c3e50;
    text-decoration: none;
}

.card a:hover {
    text-decoration: underline;
}
```

`text-decoration: none` removes the underline. The `:hover` pseudo-class applies styles only when the mouse is over the element, so the underline comes back as a visual hint that it's clickable.

One property that comes up constantly is `text-align`:

```css
h1 {
    text-align: center;
}
```

This centers the text within its container. It works on any block element. You'll use it for headings, hero sections, and footers.

Finally, spacing. You control the space between elements with `margin` (outside the element) and `padding` (inside). The difference matters once you understand the box model (next section), but the shorthand is worth knowing now:

```css
.card {
    margin: 20px 0;      /* 20px top and bottom, 0 left and right */
    padding: 20px;        /* 20px on all four sides */
}
```

You can set each side individually (`margin-top`, `padding-left`), but the shorthand covers most cases. Two values means "vertical horizontal." Four values means "top right bottom left" (clockwise from the top).

That's enough to style most of what you'll build in the next few chapters. When you need a property we haven't covered, look it up. The MDN Web Docs (developer.mozilla.org) are the best reference.

3.4 The Box Model

Every HTML element is a box. Each box has four layers, from inside to outside:

1. **Content**: The actual text, image, or other content
2. **Padding**: Space between the content and the border
3. **Border**: A line around the padding
4. **Margin**: Space between this element and neighboring elements

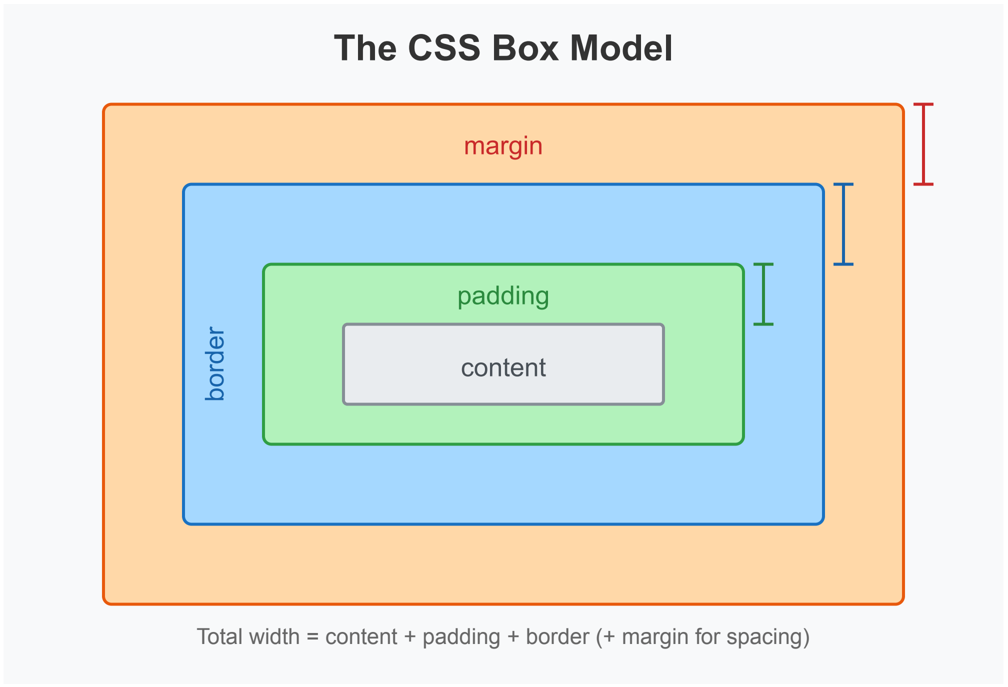

The CSS Box Model
margin
border
padding
content
Total width = content + padding + border (+ margin for spacing)

Figure 3.1: The CSS Box Model

The first three (content, padding, border) make up the element itself: its visible size on screen. Margin is different. It's not part of the element; it's the gap between this element and whatever is next to it.

This distinction matters when you set a width:

```
.card {
    width: 300px;
    padding: 20px;
    border: 2px solid black;
    margin: 10px;
}
```

By default, `width` sets only the content width. The padding and border get added on top of that. So the element's visible size on screen is:

- Content: 300px
- Padding: 20px left + 20px right = 40px
- Border: 2px left + 2px right = 4px
- Visible width: 344px

The margin adds another 10px on each side, so the element occupies 364px of horizontal space on the page. But the element itself (the box you see) is 344px wide.

This math is annoying. You set `width: 300px` and the thing ends up 344px wide. There's a fix:

```
* {
    box-sizing: border-box;
}
```

With `box-sizing: border-box`, `width` includes padding and border. Set `width: 300px` and the element is 300px wide, period. The browser subtracts space for padding and border from the content area automatically.

Most developers add this rule to every project. Put it at the top of your CSS file.

One more thing about margin: **margins collapse**. If two elements are stacked vertically and one has `margin-bottom: 20px` and the next has `margin-top: 30px`, the space between them is 30px, not 50px. The larger margin wins. This is intentional behavior, not a bug, but it surprises people. Padding never collapses.

3.5 Layout with Flexbox

By default, HTML elements stack vertically. Headings, paragraphs, divs: they all sit on top of each other, taking up the full width. That's fine for a document, but most websites need elements side by side: navigation links in a row, cards in a grid, a sidebar next to the main content.

CSS had `float` for this, which was designed for wrapping text around images and got hacked into a layout tool. It was confusing and fragile. Flexbox (Flexible Box Layout) was built specifically for the job.

To use flexbox, you set `display: flex` on a container. The container's children then become "flex items" that you can arrange:

```
<nav class="main-nav">
    <a href="/">Home</a>
    <a href="/about">About</a>
    <a href="/contact">Contact</a>
</nav>
```

```
.main-nav {
    display: flex;
}
```

That single line puts the links side by side instead of stacked. But they're crammed together with no space between them. Add `gap` to fix that:

```
.main-nav {
    display: flex;
    gap: 20px;
}
```

Now there's 20 pixels between each link. `gap` is cleaner than adding margin to each item individually because you don't have to worry about extra space on the first or last item.

The links are bunched up on the left side of the page. What if you want them spread across the full width? That's what `justify-content` controls:

```
.main-nav {
    display: flex;
    justify-content: space-between;
}
```

`space-between` pushes the first item to the left edge, the last item to the right edge, and distributes the rest evenly in between. This is the most common choice for navigation bars. If you want everything centered instead, use `justify-content: center`. If you want equal space around every item (including the edges), use `space-evenly`. You don't need to memorize all the options. `space-between` and `center` handle most cases.

Vertical alignment works similarly. If items in a row have different heights (say, an icon next to text), they might not line up the way you want. `align-items` fixes that:

```
.main-nav {
    display: flex;
    align-items: center;
}
```

This vertically centers all items within the container. Without it, items stretch to fill the container's height (the default behavior), which is sometimes what you want and sometimes not.

Centering something both horizontally and vertically on a page used to be surprisingly hard. With flexbox it's three lines:

```
.centered-container {
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;          /* full viewport height */
}
```

`justify-content` handles the horizontal axis, `align-items` handles the vertical axis. Done.

By default, flex items flow left to right in a row. Sometimes you want a vertical layout (a sidebar with stacked links, for example). `flex-direction: column` switches the axis:

```
.sidebar {
    display: flex;
    flex-direction: column;
    gap: 10px;
}
```

One thing to know: when you change direction to `column`, the roles of `justify-content` and `align-items` swap. `justify-content` now controls the vertical axis and `align-items` controls the horizontal one. This makes sense if you think of `justify-content` as "along the direction items flow" and `align-items` as "across it," but it trips people up at first.

Here's a common full-page layout that shows flexbox doing real work:

```
.page {
    display: flex;
    flex-direction: column;
    min-height: 100vh;
}

.page header {
    padding: 20px;
    background: #333;
    color: white;
}

.page main {
    flex: 1;                /* grow to fill available space */
    padding: 20px;
}

.page footer {
    padding: 20px;
    background: #333;
    color: white;
}
```

The page is a vertical flex container. Header and footer take the space they need. The `flex: 1` on `main` tells it to grow and fill whatever space is left. This keeps the footer at the bottom of the viewport even when there's not much content (a classic problem that was annoying to solve before flexbox).

3.6 Hands-On: Styling the Profile Page

Let's make the profile page from Chapter 2 look better. Create a file called `profile.css` in the same folder as your `profile.html`.

First, link the stylesheet by adding this to the `<head>` of your HTML:

```
<link rel="stylesheet" href="profile.css">
```

Now let's build the CSS step by step. Start with the basics:

```
* {
    box-sizing: border-box;
}
```

```
body {
    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, sans-serif;
    line-height: 1.6;
    color: #333;
    margin: 0;
    padding: 0;
}
```

The `font-family` line uses system fonts. The browser picks whichever one the user's operating system has. This looks native on every platform.

Style the header:

```
header {
    background-color: #2c3e50;
    color: white;
    padding: 40px 20px;
    text-align: center;
}

header h1 {
    margin: 0 0 10px 0;
    font-size: 2.5em;
}

header nav {
    display: flex;
    justify-content: center;
    gap: 20px;
}

header nav a {
    color: white;
    text-decoration: none;
    padding: 5px 10px;
}

header nav a:hover {
    text-decoration: underline;
}
```

The `:hover` pseudo-class applies when the mouse is over the element.

Style the main content:

```
main {
    max-width: 800px;
    margin: 0 auto;
    padding: 40px 20px;
}

section {
    margin-bottom: 40px;
}

h2 {
    color: #2c3e50;
    border-bottom: 2px solid #3498db;
    padding-bottom: 10px;
}
```

`margin: 0 auto` centers a block element horizontally (the `auto` margins on left and right split the remaining space equally).

Now it's your turn. Using what you've learned, style these remaining elements:

The skills list should display as horizontal tags instead of bullet points. Remove the default list styling, use flexbox to arrange the items in a row, and make each skill look like a pill (background color, padding, rounded corners). Allow items to wrap if they don't fit on one line.

The contact form should have its fields stacked vertically with some space between them. Style the inputs and textarea with padding, a border, and rounded corners. Make the submit button stand out with a background color, and add a hover effect so users know it's clickable.

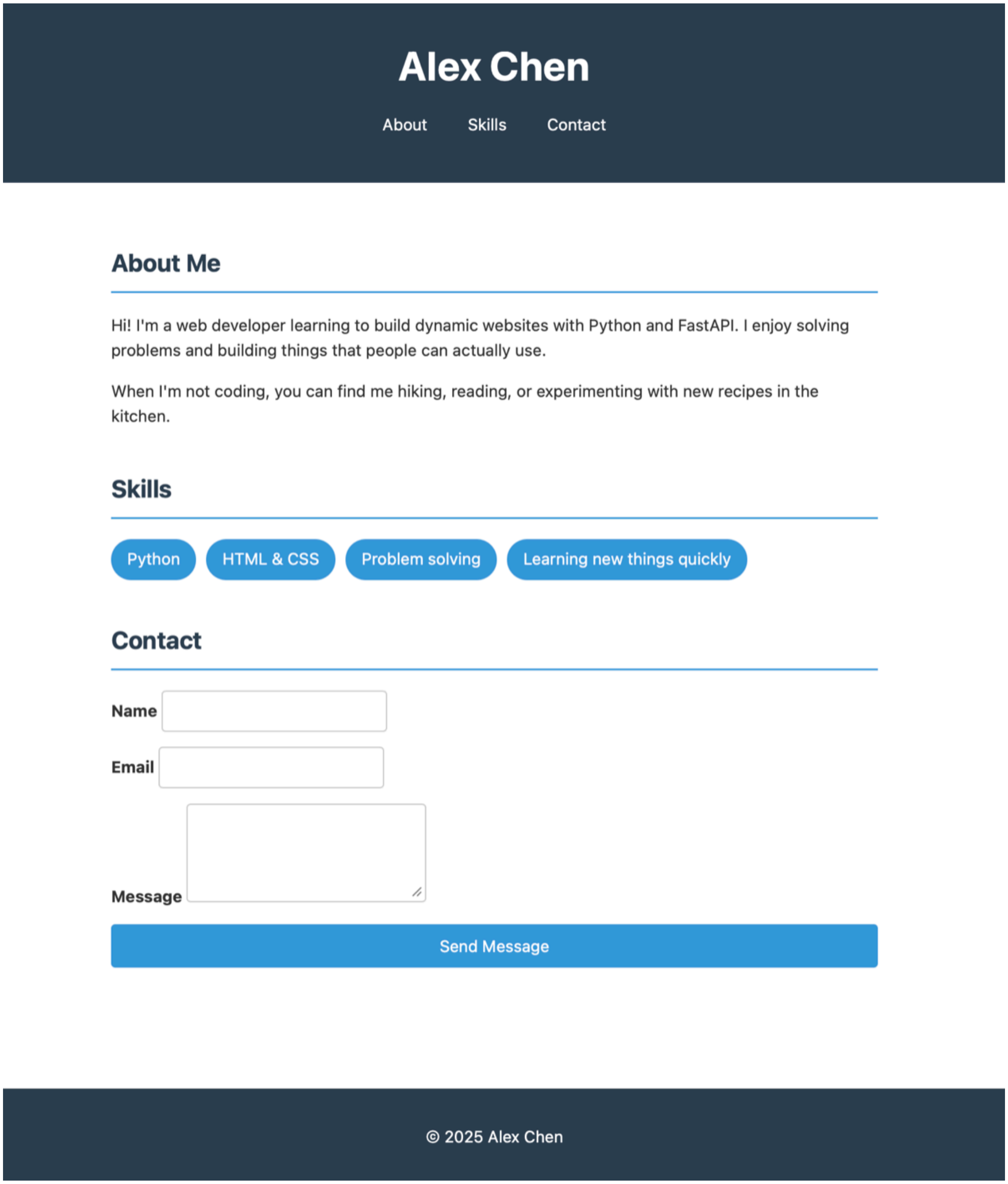
The footer should match the header's style (same background color, white text, centered).

Experiment and check your results in the browser as you go. If you get stuck, the `:focus` pseudo-class is useful for styling form fields when they're selected, and `cursor: pointer` on buttons tells the browser to show the pointing hand cursor.

Stuck? Solution available

If you've given it an honest try and are truly stuck, a completed version of this project is available at github.com/Applied-Computing-League/practical-web-development/en/code/chapter3. Try to solve it yourself first. You learn more from the struggle than from reading the answer.

When you're done, your page should look something like this:



About Me

Hi! I'm a web developer learning to build dynamic websites with Python and FastAPI. I enjoy solving problems and building things that people can actually use.

When I'm not coding, you can find me hiking, reading, or experimenting with new recipes in the kitchen.

Skills

Contact

Figure 3.2: The styled profile page

Try resizing your browser window. The layout should adapt reasonably well. The header stays centered, the content has a maximum width, and the skills wrap to new lines.

3.7 Chapter Summary

- CSS controls how HTML elements look: colors, fonts, spacing, layout
- Connect CSS to HTML with `<link rel="stylesheet" href="file.css">`
- Selectors target elements: `p` (element), `.class` (class), `#id` (ID)
- The box model: content + padding + border + margin
- Use `box-sizing: border-box` to make width calculations simpler
- Flexbox handles layout: `display: flex`, `justify-content`, `align-items`, `gap`

We now have a styled page, but it's still static. In the next chapter, we'll start building a server that can generate pages dynamically.

3.8 Exercises

1. Your profile page uses the color #2c3e50 in multiple places (header, footer, headings). What happens if you want to change it? Count how many places you'd need to update. Then look up CSS custom properties (variables) and refactor your CSS so the color is defined once and referenced everywhere. Change it to a new color by editing a single line.
2. Add a hover effect to the skill tags. Maybe they change color, grow slightly, or get a shadow. Look up `transform: scale()` and `box-shadow`.
3. Make the navigation links look like buttons (background color, padding, rounded corners). Then add two states: a hover effect that changes the background, and a style that visually marks the "current" page. You'll need to add a class to one of the nav links in your HTML and write a CSS rule for it.
4. Add a profile image (real or placeholder) to the header. Center it above your name. Make it circular using `border-radius: 50%`.
5. Create a "projects" section with three project cards side by side using flexbox. Each card should have a title, description, and a "View Project" link. Make the cards equal width (look up flex: 1). Then resize your browser window — what happens when the cards get too narrow? Look up flex-wrap and set a min-width on each card so they wrap to a new row on smaller screens.
6. Using developer tools, inspect your styled page. Find an element and try changing its CSS values in the inspector. Watch how the page updates in real time. This is useful for experimenting before changing your actual CSS file.

4 Chapter 4: Servers with FastAPI

So far we've written HTML and CSS files and opened them directly in a browser. That works, but those are static files. The browser reads them from your computer and displays them. No server involved.

In Chapter 1, we talked about dynamic websites: servers that run code to generate responses. Now we're going to build one. When someone visits your site, your code will run, decide what to send back, and return it. The browser won't be reading files from disk. It'll be making HTTP requests to your server, and your server will respond.

We're using Python with a framework called FastAPI. A framework is a collection of code that handles the boring parts (parsing HTTP requests, routing URLs, sending responses) so you can focus on the interesting parts (what your application actually does). FastAPI is modern, fast enough, and has good documentation. It's also relatively light, which is a pro for us. Since the main goal is learning, we want to understand what's happening rather than have magic do everything.

4.1 What a Server Actually Does

A web server's job is simple: wait for HTTP requests, then send HTTP responses. That's it.

When you run a server on your computer, it listens on a port (usually 8000 for development). When a browser makes a request to `http://localhost:8000/about`, your server receives that request, sees that the path is `/about`, runs whatever code you've associated with that path, and sends back the result.

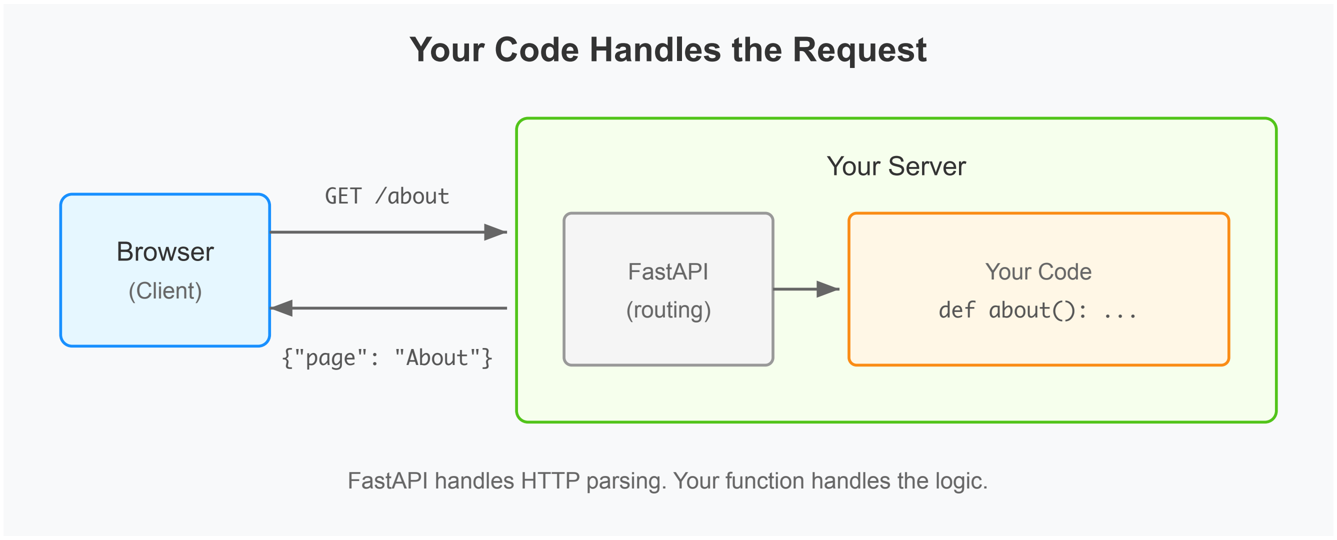


Figure 4.1: Your code handles the request

The code you write defines what happens for each path. Visit `/`? Run this function. Visit `/about`? Run that function. Visit `/users/42`? Run another function and pass it the number 42. This mapping from paths to functions is called **routing**.

FastAPI handles the HTTP parsing and response formatting. You just write Python functions that return data, and FastAPI turns that data into proper HTTP responses.

4.2 Setting Up Your Project

Create a new folder for this chapter:

```
mkdir chapter4
cd chapter4
```

We'll use `uv` to manage our Python environment and dependencies. If you don't have `uv` installed:

```
# On macOS/Linux
curl -LsSf https://astral.sh/uv/install.sh | sh

# On Windows
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Now initialize a new project and add FastAPI:

```
uv init
uv add "fastapi[standard]"
```

The `[standard]` part installs FastAPI along with useful extras, including the server that will run our code.

This creates a few files. Delete the `hello.py` that `uv init` creates (we don't need it), and make a new file called `main.py`. Your folder should look like this:

```
chapter4/
├── .venv/
├── .python-version
├── pyproject.toml
└── main.py
```

And these are the files we'll populate.

4.3 Type Hints

Before we write our first server, we need to talk about type hints. FastAPI uses them heavily, and they're worth understanding.

Type hints let you declare what type a variable or parameter should be:

```python
def greet(name: str) -> str:
    return f"Hello, {name}!"
```

The `name: str` says this function expects a string. The `-> str` says it returns a string. Python doesn't enforce these at runtime, but your editor uses them to catch mistakes and provide better autocomplete.

FastAPI uses type hints for something more powerful: automatic validation and conversion. When you write `user_id: int` in a FastAPI function, FastAPI will automatically convert the incoming string from the URL to an integer, and return an error if it can't. You get validation for free.

You can also indicate that something might be missing:

```python
def greet(name: str | None = None) -> str:
    if name:
        return f"Hello, {name}!"
    return "Hello, stranger!"
```

The `str | None` means "either a string or None." The `= None` makes it optional with a default value. That's enough type hint knowledge to get started. You'll pick up more as we go.

4.4 Your First Server

Now let's write some code. In `main.py`, start with the import and app creation:

```python
from fastapi import FastAPI

app = FastAPI()
```

`from fastapi import FastAPI` imports the FastAPI class. `app = FastAPI()` creates your application. This object handles all the HTTP stuff.

Now add your first route:

```python
# main.py
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def home():
    return {"message": "Hello, world!"}
```

`@app.get("/")` is a decorator that tells FastAPI: when someone makes a GET request to `/`, run the function below.

`def home()` is a regular Python function. It returns a dictionary, and FastAPI automatically converts that to JSON and sends it to the browser.

To run the server:

```
uv run fastapi dev main.py
```

You'll see output indicating the server is running at `http://127.0.0.1:8000`. Open that URL in your browser. You should see:

```
{"message": "Hello, world!"}
```

That's JSON. Your server received an HTTP request and sent back an HTTP response, just like we discussed in Chapter 1. Open your browser's dev tools (F12), go to the Network tab, and refresh. You can see the request your browser made and the response your server sent.

The `fastapi dev` command watches for changes and automatically restarts when you edit your code. Keep it running while you work through this chapter.

4.5 Routes and Endpoints

A **route** is a path that your server responds to. An **endpoint** is the function that handles that route. Let's add more routes. Add these below your existing `home` function:

```python
# main.py
# ...existing code above

@app.get("/about")
def about():
    return {"page": "About", "description": "This is the about page."}


@app.get("/contact")
def contact():
    return {"email": "hello@example.com"}
```

Save the file. The server restarts automatically. Now try these URLs:

- `http://localhost:8000/` returns the hello message
- `http://localhost:8000/about` returns about info
- `http://localhost:8000/contact` returns contact info
- `http://localhost:8000/anything-else` returns a 404 error

Each `@app.get()` decorator registers a route. The path in the decorator determines which URL triggers which function. Here's what that mapping looks like:

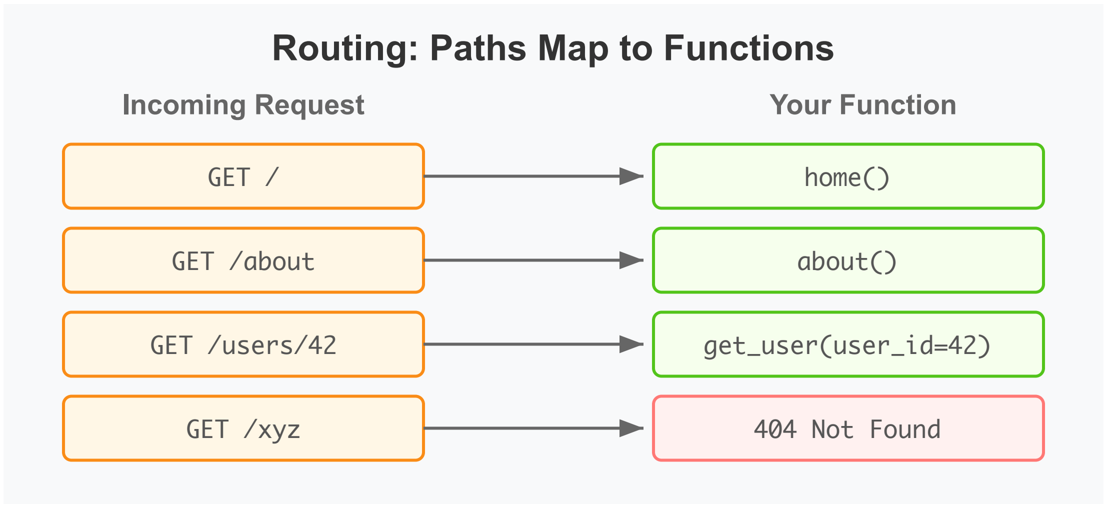


Figure 4.2: Routing maps paths to functions

The function names don't matter to FastAPI (you could call them all `def x():`), but good names make your code more readable and help with automatic documentation (we'll see more about this later).

4.6 Path Parameters

Sometimes you want part of the URL to be a variable. Consider a user profile page: you don't want to write a separate route for every user. You want one route that works for any user ID.

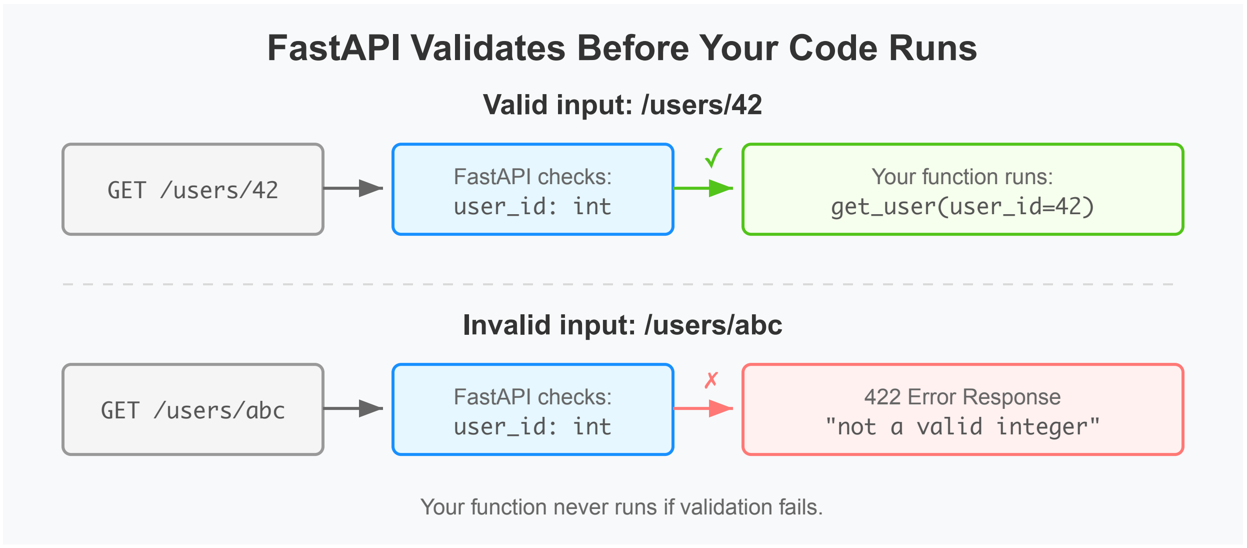
```
# main.py
# ...existing code above

@app.get("/users/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id, "name": f"User {user_id}"}
```

The `{user_id}` in the path is a **path parameter**. FastAPI extracts it from the URL and passes it to your function. The `user_id: int` type hint tells FastAPI to convert it to an integer.

Add it to your FastAPI application, then try these:

- `http://localhost:8000/users/1` returns `{"user_id": 1, "name": "User 1"}`
- `http://localhost:8000/users/42` returns `{"user_id": 42, "name": "User 42"}`
- `http://localhost:8000/users/abc` returns an error because "abc" isn't a valid integer

That last one is interesting. You didn't write any validation code, but FastAPI returns a clear error message when the input is wrong. Here's what's happening behind the scenes:



FastAPI Validates Before Your Code Runs
Valid input: /users/42
GET /users/42
FastAPI checks: user_id: int
✓
Your function runs: get_user(user_id=42)
Invalid input: /users/abc
GET /users/abc
FastAPI checks: user_id: int
✗
422 Error Response "not a valid integer"
Your function never runs if validation fails.

Figure 4.3: FastAPI validates before your code runs

That's the type hints doing their job. FastAPI checks the input, and if it doesn't match, your function never even runs.

You can have multiple path parameters:

```
@app.get("/users/{user_id}/posts/{post_id}")
def get_user_post(user_id: int, post_id: int):
    return {"user_id": user_id, "post_id": post_id}
```

This pattern is common for nested resources: a post belongs to a user, so the URL reflects that relationship. The order of parameters in your function doesn't matter, FastAPI matches them by name. Path parameters work when the value identifies a specific resource (/users/42). But what about optional filters or settings, like limiting search results or filtering by category?

4.7 Query Parameters

Path parameters are part of the URL path. **Query parameters** come after the `?` in a URL, like `/search?q=python&limit=10`. Remember the URL anatomy from Chapter 1?

In FastAPI, any function parameter that isn't in the path is treated as a query parameter:

```
# main.py
# ...existing code above

@app.get("/search")
def search(q: str, limit: int = 10):
    return {"query": q, "limit": limit}
```

Add this to your code, then try these:

- `http://localhost:8000/search?q=python` returns `{"query": "python", "limit": 10}`
- `http://localhost:8000/search?q=python&limit=5` returns `{"query": "python", "limit": 5}`
- `http://localhost:8000/search` returns an error because `q` is required

The `limit: int = 10` default value makes that parameter optional. If the user doesn't provide it, it defaults to 10 (just like in a regular Python function).

You can make a parameter optional with no default by using `None`:

```
@app.get("/items")
def list_items(category: str | None = None):
    if category:
        return {"filter": category, "items": []}
    return {"items": []}
```

You've probably noticed that all our endpoints return dictionaries, and FastAPI converts them to JSON. That's not an accident. We've been building an API.

4.8 What is an API?

You've probably heard the term "API" before. It stands for Application Programming Interface. The key word is "interface": it's a contract between two pieces of software that defines how they can talk to each other.

Think of it like a language. When you call a function in Python, you need to know its name, what arguments it takes, and what it returns. That's an interface. A web API is the same idea, but over HTTP: it defines what URLs exist, what parameters they accept, and what data they return.

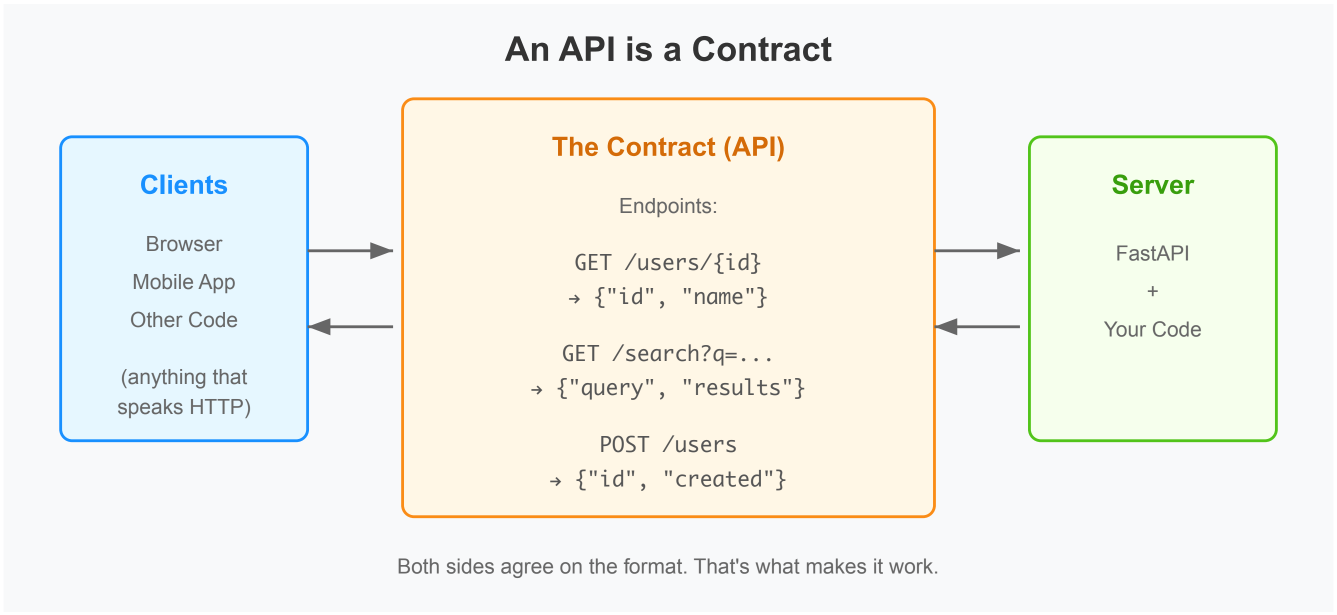

An API is a Contract
Clients
Browser
Mobile App
Other Code
(anything that speaks HTTP)
The Contract (API)
Endpoints:
GET /users/{id} → {"id", "name"}
GET /search?q=... → {"query", "results"}
POST /users → {"id", "created"}
Server
FastAPI
+
Your Code
Both sides agree on the format. That's what makes it work.

Figure 4.4: An API is a contract

In web development, "API" usually means a server that returns structured data (often JSON) instead of HTML pages. When you visit a website, you're a human looking at HTML. When an app on your phone fetches your tweets or a JavaScript frontend fetches product data, it's code talking to an API. The API returns data in a format that code can easily parse and use.

FastAPI is designed for building APIs (it's in the name), but it can also return HTML. We'll do both in this book.

4.9 Returning HTML

So far, all our endpoints return dictionaries, and FastAPI converts them to JSON. That's useful when your server is talking to other code (a mobile app, a JavaScript frontend, another server). But when a human visits your site in a browser, they don't want to see raw JSON. They want a rendered page.

The good news: your server can do both. Some endpoints return JSON for code to consume, others return HTML for humans to read. Many real applications do exactly this. The JSON endpoints are your API; the HTML endpoints are your website. Same server, different responses based on what's being asked for.

To return HTML, import `HTMLResponse`:

```
from fastapi import FastAPI
from fastapi.responses import HTMLResponse

app = FastAPI()
```

Then use it on an endpoint:

```
# main.py
# ...existing code above

@app.get("/page", response_class=HTMLResponse)
def page():
    return """
    <!DOCTYPE html>
    <html>
    <head>
        <title>My Site</title>
    </head>
    <body>
        <h1>Welcome!</h1>
        <p>This is my website.</p>
    </body>
    </html>
    """
```

The `response_class=HTMLResponse` tells FastAPI to set the `Content-Type` header to `text/html` instead of `application/json`. The browser sees HTML and renders it as a page instead of showing raw text.

You can have JSON and HTML endpoints in the same application. Your `/users/{user_id}` endpoint from earlier still returns JSON. This new `/page` endpoint returns HTML. The server doesn't care; it sends whatever your function returns with the right headers.

But writing HTML inside Python strings gets messy fast. Look at that function: it's a Python function that's mostly HTML. Imagine adding a navigation bar, a form, conditional content based on user data. The string would grow to hundreds of lines, your editor couldn't help with the HTML (no syntax highlighting, no autocomplete inside strings), and debugging a missing closing tag would be painful.

We'll solve this properly in the next chapter with templates. For now, let's use this approach to serve something real and see the payoff of generating HTML dynamically.

4.10 Hands-On: Serving the Profile Page

Remember the profile page from Chapters 2 and 3? We opened it directly in the browser as a file. Now let's serve it from a real server. We'll start a fresh `main.py` for this, since we're building a different application than the test endpoints above.

First, create a `static` folder for our CSS:

```
chapter4/
├── .venv/
├── main.py
├── pyproject.toml
└── static/
    └── profile.css
```

Copy your `profile.css` from Chapter 3 into the `static` folder.

Now let's build `main.py`. Start fresh with the imports:

```
from fastapi import FastAPI
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles

app = FastAPI()
```

Next, tell FastAPI where to find static files:

```
# main.py
# ...imports and app above

app.mount("/static", StaticFiles(directory="static"), name="static")
```

This line tells FastAPI: any request starting with `/static/` should look for files in the `static` folder. This is how web applications serve CSS, JavaScript, and images.

Now add the profile data:

```
# main.py
# ...static files setup above

profile = {
    "name": "Philipe Ackerman",
    "about": [
        "Hi! I'm a math professor, and a really good one at that.",
        "When I'm not teaching, you can find me cheering for my soccer team, Botafogo.",
    ],
    "skills": ["Math", "Having Long Hair", "Video Recording"],
}
```

This dictionary is our data. Philipe is a math professor who teaches algebra and geometry on YouTube, pointer stick in hand, long hair and all. Right now his profile is hardcoded, but it could come from a database, a file, or user input. The point is that the HTML is generated from data, and when the data changes, the page changes with it.

Finally, the endpoint that generates the HTML. First, we build the dynamic parts:

```
# main.py
# ...profile data above

@app.get("/", response_class=HTMLResponse)
def home():
    skills_html = "".join(f"<li>{skill}</li>" for skill in profile["skills"])
    about_html = "".join(f"<p>{para}</p>" for para in profile["about"])
```

We loop through the skills and about paragraphs to generate the list items and paragraphs as HTML strings.

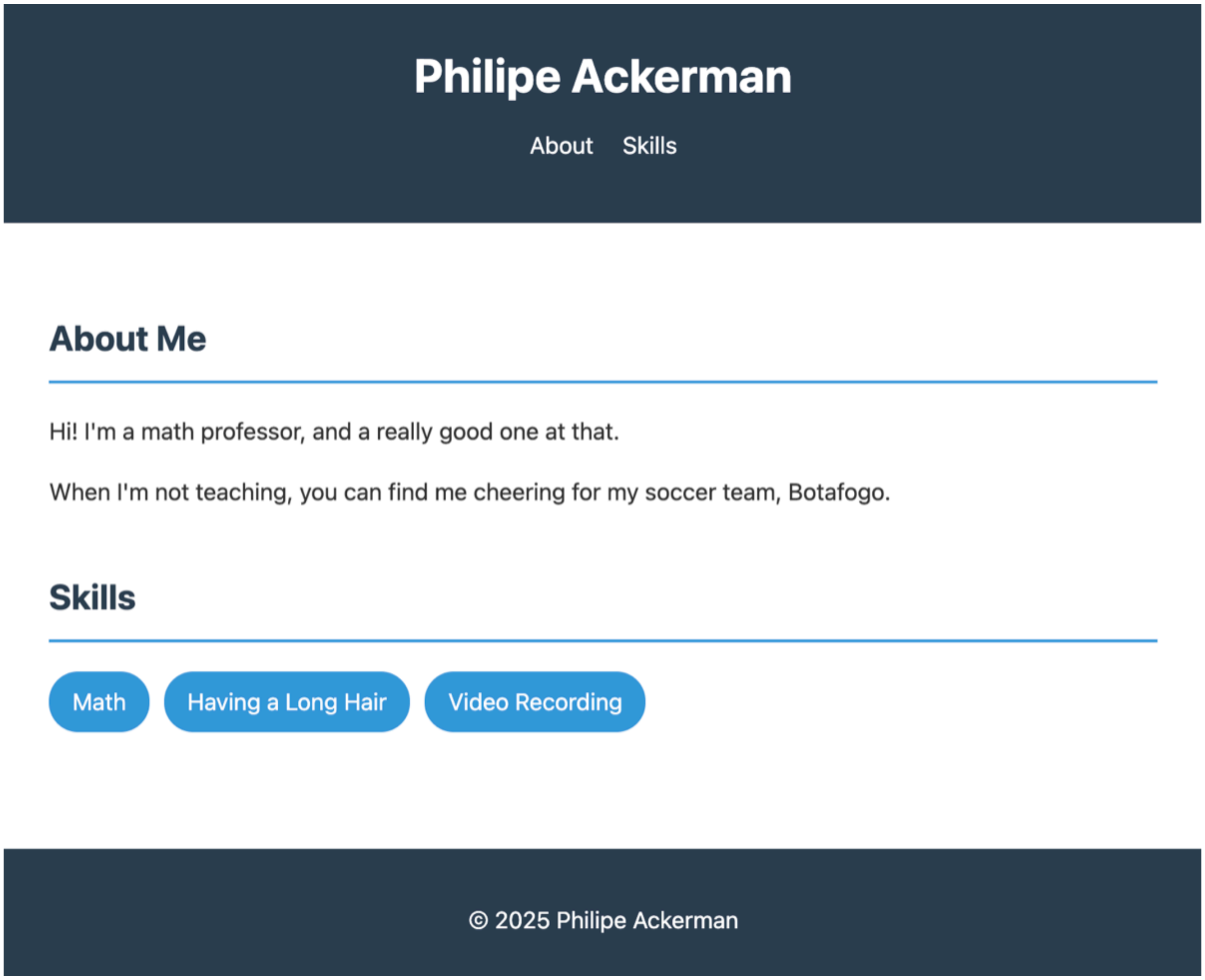
Then we return the full HTML page using an f-string:

```
# main.py
# inside the home() function, after building skills_html and about_html

    return f"""
    <!DOCTYPE html>
    <html>
    <head>
        <title>{profile["name"]} - Profile</title>
        <link rel="stylesheet" href="/static/profile.css">
    </head>
    <body>
        <header>
            <h1>{profile["name"]}</h1>
            <nav>
                <a href="#about">About</a>
                <a href="#skills">Skills</a>
            </nav>
        </header>

        <main>
            <section id="about">
                <h2>About Me</h2>
                {about_html}
            </section>

            <section id="skills">
                <h2>Skills</h2>
                <ul>
                    {skills_html}
                </ul>
            </section>
        </main>

        <footer>
            <p>&copy; 2025 {profile["name"]}</p>
        </footer>
    </body>
    </html>
    """
```

The f-string inserts our profile data into the HTML. Change the name in the dictionary, save, and refresh.

The page updates. This is the core idea of dynamic websites: the same code generates different pages based on different data.

Run the server and visit `http://localhost:8000`. You should see the styled profile page:


Philipe Ackerman

About Skills

About Me

Hi! I'm a math professor, and a really good one at that.

When I'm not teaching, you can find me cheering for my soccer team, Botafogo.

Skills

© 2025 Philipe Ackerman

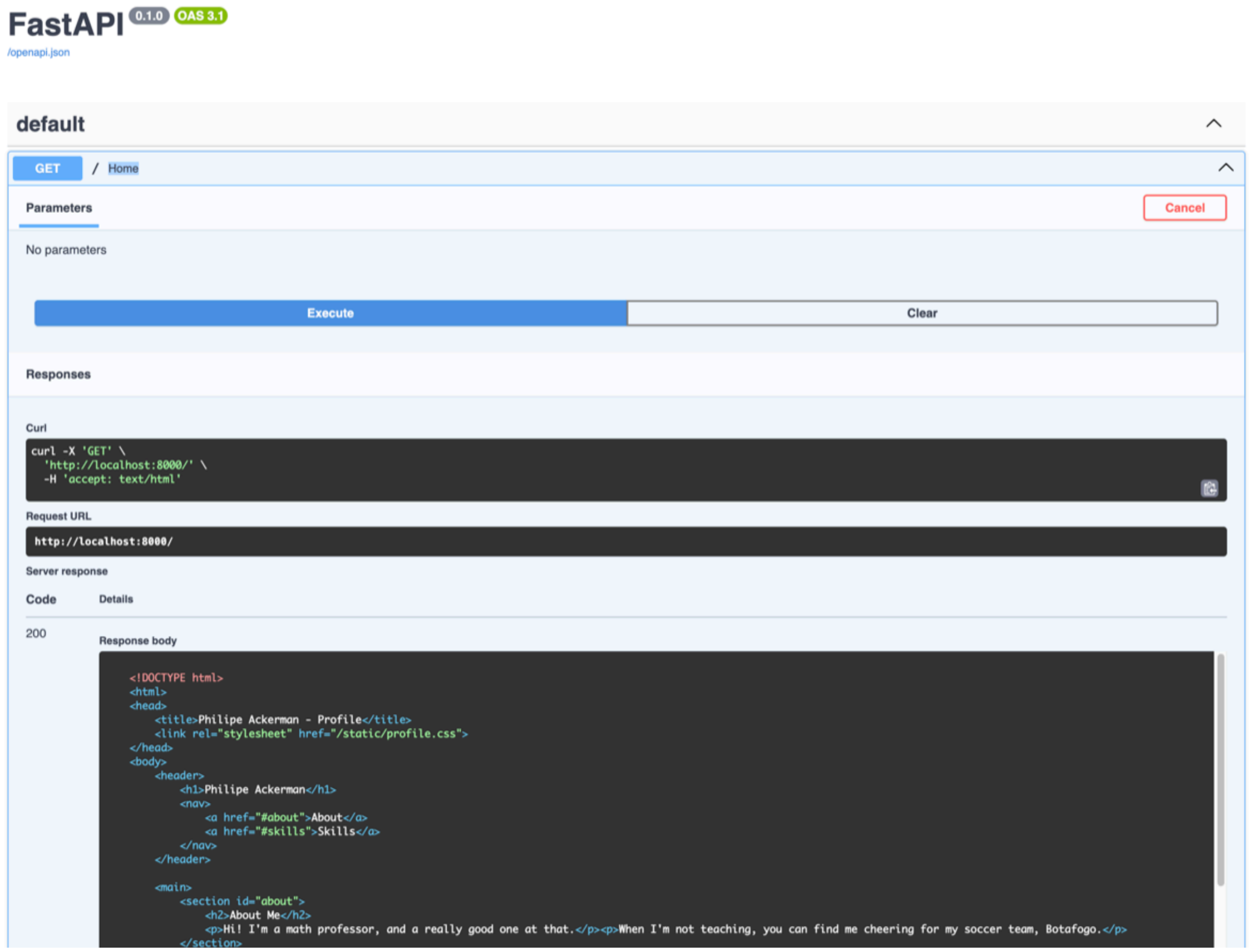
Figure 4.5: The profile page served by FastAPI

Look at the Network tab in your dev tools. You'll see two requests: one for the HTML page and one for the CSS file.

The HTML-in-Python-strings approach is getting unwieldy though. Imagine a page with forms, multiple sections, and conditional content. In the next chapter, we'll use templates to separate our HTML from our Python code properly.

4.11 Automatic API Documentation

Before we wrap up, there's one more FastAPI feature worth seeing. Visit `http://localhost:8000/docs` while your server is running:



```
curl -X 'GET' \
  'http://localhost:8000/' \
  -H 'accept: text/html'
```

```
http://localhost:8000/
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Philipe Ackerman - Profile</title>
    <link rel="stylesheet" href="/static/profile.css">
</head>
<body>
    <header>
        <h1>Philipe Ackerman</h1>
        <nav>
            <a href="#about">About</a>
            <a href="#skills">Skills</a>
        </nav>
    </header>

    <main>
        <section id="about">
            <h2>About Me</h2>
            <p>Hi! I'm a math professor, and a really good one at that.</p><p>When I'm not teaching, you can find me cheering for my soccer team, Botafogo.</p>
        </section>
```

Figure 4.6: FastAPI generates interactive documentation

FastAPI automatically generates interactive documentation for all your endpoints. It shows the paths, the parameters, the expected types, and even lets you test the endpoints directly from the browser. This is generated from your code (the function names, type hints, and docstrings all contribute).

This documentation is useful when you're building an API that other developers will use. They can see exactly what endpoints exist and what data they expect, without reading your code. It's also helpful during development: you can test your endpoints without writing any client code.

4.12 Chapter Summary

- A web server waits for HTTP requests and sends responses
- FastAPI handles the HTTP details; you write Python functions
- Type hints tell FastAPI what types to expect and it validates automatically
- Routes map URL paths to functions: `@app.get("/path")`
- Path parameters are variables in the URL: `/users/{user_id}`
- Query parameters come after `?`: `/search?q=term&limit=10`
- Return a dict for JSON; use `HTMLResponse` for HTML
- Use `StaticFiles` to serve CSS, JavaScript, and images
- Run with `uv run fastapi dev main.py`
- Visit `/docs` for automatic API documentation

Writing HTML in Python strings is awkward. In the next chapter, we'll use templates to keep HTML in separate files where it belongs.

4.13 Exercises

1. Add an `/api/profile` endpoint that returns the profile data as JSON instead of HTML. Visit it in your browser and compare the `Content-Type` headers between `/` and `/api/profile` using dev tools.
2. Add a path parameter to create different profiles: `/profile/{username}`. Create a dictionary with a few different profiles and return the matching one. If the username isn't found, return a 404 error (look up `HTTPException` in the FastAPI documentation).
3. Add a `/api/skills` endpoint that returns just the skills list as JSON. Then add an optional `limit` query parameter that limits how many skills are returned.
4. Add a "contact" section to the profile page. Store an email address in the profile dictionary and display it on the page.
5. Open `/docs` and try out your endpoints from there. Add a docstring to one of your endpoint functions and see how it appears in the documentation.
6. Using your browser's dev tools, watch the Network tab while you refresh the profile page. How many requests does the browser make? What happens if the CSS file is missing?

5 Chapter 5: Templates with Jinja

At the end of Chapter 4, we had HTML inside a Python string. It worked, but it was ugly. The string got long, the indentation was awkward, and mixing Python with HTML made both harder to read. There's a better way.

5.1 Why Not Write HTML in Python?

Look at the code we ended up with:

```python
@app.get("/", response_class=HTMLResponse)
def home():
    skills_html = "".join(f"<li>{skill}</li>" for skill in profile["skills"])
    about_html = "".join(f"<p>{para}</p>" for para in profile["about"])

    return f"""
    <!DOCTYPE html>
    <html>
    <head>
        <title>{profile["name"]} - Profile</title>
        ...
    """
```

This has problems. Your editor can't help you with HTML inside a string: no syntax highlighting, no autocomplete, no error checking. If you forget a closing tag, you won't know until you see broken output in the browser. And as pages get more complex, these strings become unmanageable.

The solution is templates. A template is an HTML file with placeholders for dynamic content. You write your HTML in separate files where your editor can help you, mark where the data should go, and let a template engine fill in the blanks.

We'll use Jinja, the most popular Python template engine. It's already installed with FastAPI.

5.2 Template Basics

Create a new folder for this chapter and set it up:

```bash
mkdir chapter5
cd chapter5
```

```
uv init
uv add "fastapi[standard]"
```

Now create a `templates` folder:

```
chapter5/
├── .venv/
├── main.py
├── pyproject.toml
└── templates/
    └── home.html
```

In `templates/home.html`, write a simple template:

```html
<!DOCTYPE html>
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

The { } syntax marks a placeholder. When the template renders, { `title` } gets replaced with an actual value. This is just HTML with holes in it.

Now set up FastAPI to use templates. In `main.py`:

```python
from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates

app = FastAPI()
templates = Jinja2Templates(directory="templates")
```

The `Jinja2Templates` object knows where to find your template files. Now create a route that uses a template:

```python
# main.py
# ...imports and setup above

@app.get("/")
def home(request: Request):
    return templates.TemplateResponse(
        request=request,
        name="home.html",
        context={"title": "My Site", "name": "World"}
    )
```

A few things to note. The function takes a `request` parameter. FastAPI provides this automatically, and we pass it to `TemplateResponse` because Jinja needs it for certain features.

The `name` is which template file to use. The `context` is a dictionary of values to fill in the placeholders. `{ title }` becomes "My Site" and `{ name }` becomes "World".

Here's what happens when someone visits your page:

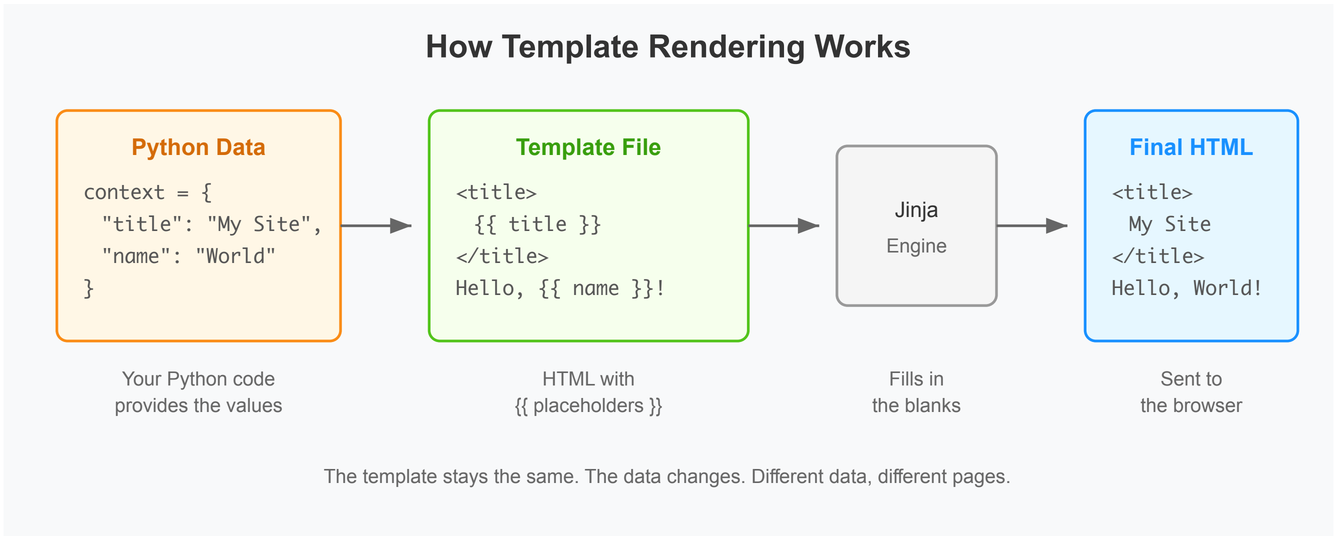


Figure 5.1: How template rendering works

Run the server with `uv run fastapi dev main.py` and visit `http://localhost:8000`. You should see "Hello, World!" with the page title "My Site". The HTML came from the template file, with the placeholders replaced by our data.

5.3 Passing Data to Templates

The context dictionary can contain anything: strings, numbers, lists, dictionaries, even objects. Let's pass more interesting data:

```
# main.py
# ...imports and setup above

@app.get("/")
def home(request: Request):
    user = {
        "name": "Philipe",
        "role": "Math Professor",
        "active": True
    }
    return templates.TemplateResponse(
        request=request,
        name="home.html",
        context={"title": "Profile", "user": user}
    )
```

In the template, you access dictionary keys with dot notation:

```
<!DOCTYPE html>
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ user.name }}</h1>
    <p>Role: {{ user.role }}</p>
</body>
</html>
```

`{ user.name }` reaches into the `user` dictionary and gets the `name` key. This works the same whether `user` is a dictionary or a Python object with attributes.

5.4 Loops and Conditionals in Templates

Templates can do more than insert values. You can loop over lists and make decisions based on conditions.

To display a list of items, use `{% for %}`. Update your template:

```
<h2>Skills</h2>
<ul>
{% for skill in skills %}
    <li>{{ skill }}</li>
{% endfor %}
</ul>
```

The `{% %}` syntax is for logic (loops, conditionals), while `{ }` is for outputting values. Everything between `{% for %}` and `{% endfor %}` repeats for each item in the list.

Pass the list from Python:

```
context = {
    "title": "Profile",
    "user": user,
    "skills": ["Python", "HTML", "CSS", "FastAPI"]
}
```

The template produces:

```
<h2>Skills</h2>
<ul>
    <li>Python</li>
    <li>HTML</li>
    <li>CSS</li>
    <li>FastAPI</li>
</ul>
```

You can also loop over lists of dictionaries, which is common when displaying data from a database:

```
{% for project in projects %}
    <div>
        <h3>{{ project.name }}</h3>
        <p>{{ project.description }}</p>
    </div>
{% endfor %}
```

Conditionals work similarly. To show content only when a condition is true, use `{% if %}`:

```
{% if user.active %}
    <span class="badge">Active</span>
{% endif %}
```

You can add `{% else %}` for the alternative:

```
{% if user.active %}
    <span class="badge active">Active</span>
{% else %}
    <span class="badge inactive">Inactive</span>
{% endif %}
```

And `{% elif %}` for multiple conditions:

```
{% if user.role == "admin" %}
    <span class="badge admin">Admin</span>
{% elif user.role == "moderator" %}
    <span class="badge mod">Moderator</span>
{% else %}
    <span class="badge">Member</span>
{% endif %}
```

Jinja supports the comparison operators you'd expect: ==, !=, <, >, <=, >=. You can combine conditions with `and`, `or`, and `not`.

5.5 Template Inheritance

Most websites have consistent elements on every page: the same header, the same footer, the same navigation. Copying that HTML into every template would be tedious and error-prone. Template inheritance solves this.

You create a base template that defines the common structure, then child templates that fill in the parts that change.

Create `templates/base.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My Site{% endblock %}</title>
    <link rel="stylesheet" href="/static/style.css">
</head>
<body>
    <header>
        <h1>My Site</h1>
        <nav>
            <a href="/">Home</a>
            <a href="/about">About</a>
        </nav>
    </header>

    <main>
        {% block content %}{% endblock %}
    </main>

    <footer>
        <p>&copy; 2025 My Site</p>
    </footer>
</body>
</html>
```

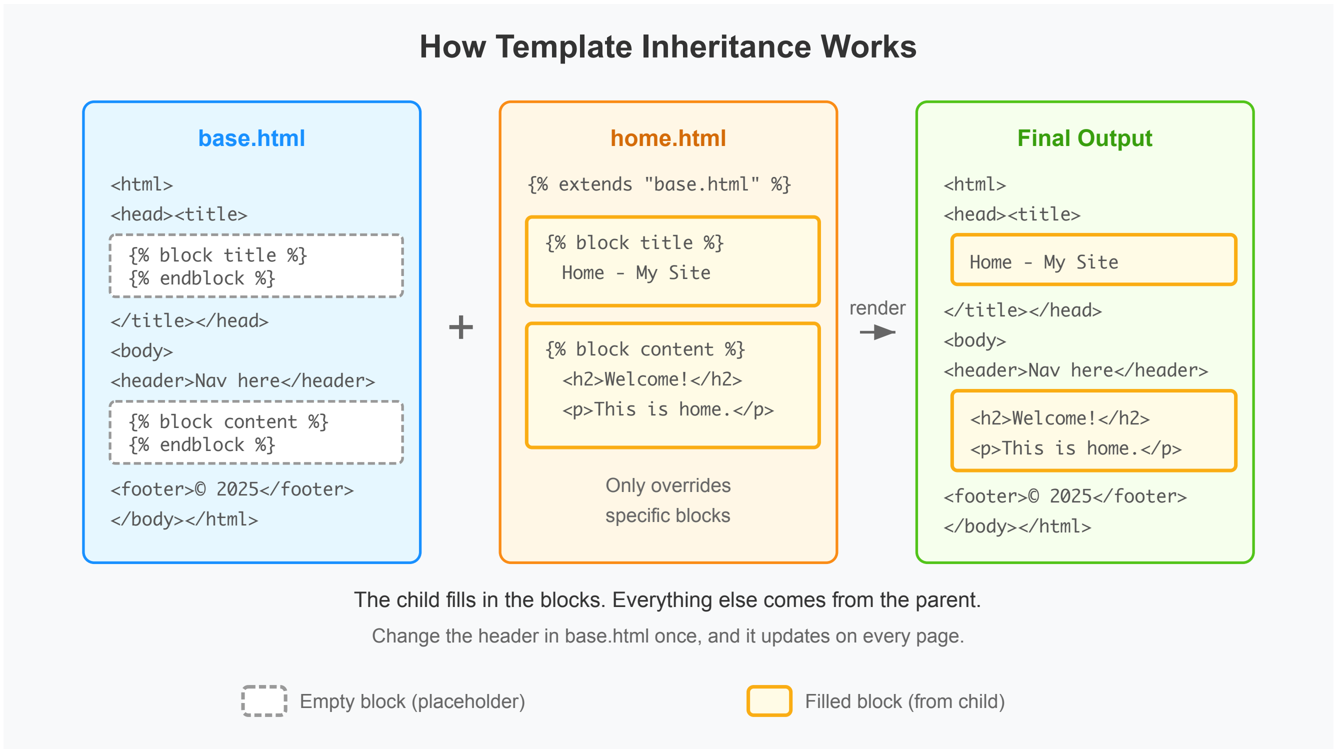
The `{% block %}` tags define holes that child templates can fill. The text inside (`My Site` in the title block) is the default if the child doesn't override it.

Now create a child template. Update `templates/home.html`:

```
{% extends "base.html" %}

{% block title %}Home - My Site{% endblock %}

{% block content %}
<h2>Welcome!</h2>
<p>This is the homepage.</p>
{% endblock %}
```

`{% extends "base.html" %}` says this template builds on base.html. The `{% block %}` tags override the parent's blocks. Everything else comes from the parent.

Here's how the pieces fit together:


How Template Inheritance Works
base.html
home.html
Final Output
+
render
Only overrides specific blocks
The child fills in the blocks. Everything else comes from the parent.
Change the header in base.html once, and it updates on every page.
Empty block (placeholder)
Filled block (from child)

```
<html>
<head><title>
  {% block title %}
  {% endblock %}
</title></head>
<body>
<header>Nav here</header>
  {% block content %}
  {% endblock %}
<footer>© 2025</footer>
</body></html>
```

```
{% extends "base.html" %}
  {% block title %}
    Home - My Site
  {% block content %}
    <h2>Welcome!</h2>
    <p>This is home.</p>
```

```
<html>
<head><title>
  Home - My Site
</title></head>
<body>
<header>Nav here</header>
  <h2>Welcome!</h2>
  <p>This is home.</p>
<footer>© 2025</footer>
</body></html>
```

Figure 5.2: How template inheritance works

Create another page to see the benefit. Add `templates/about.html`:

```
{% extends "base.html" %}

{% block title %}About - My Site{% endblock %}

{% block content %}
<h2>About Us</h2>
<p>We make things.</p>
{% endblock %}
```

And add the route in `main.py`:

```
# main.py
# ...existing code above

@app.get("/about")
def about(request: Request):
    return templates.TemplateResponse(
        request=request,
        name="about.html",
        context={}
    )
```

Both pages share the same header, navigation, and footer. Change the nav in `base.html` and it changes everywhere. This is why real websites use template inheritance.

5.6 Hands-On: Converting the Profile Page

Let's convert our profile page from Chapter 4 to use templates properly. First, set up the folder structure:

```
chapter5/
├── .venv/
├── main.py
├── pyproject.toml
├── static/
│   └── profile.css
└── templates/
    ├── base.html
    └── profile.html
```

Copy your `profile.css` from Chapter 4 into the `static` folder.

Start with `templates/base.html`. This will be our foundation for all pages:

```html
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Profile{% endblock %}</title>
    <link rel="stylesheet" href="/static/profile.css">
</head>
<body>
    <header>
        <h1>{% block header_title %}{% endblock %}</h1>
        <nav>
            <a href="#about">About</a>
            <a href="#skills">Skills</a>
            <a href="#contact">Contact</a>
        </nav>
    </header>

    <main>
        {% block content %}{% endblock %}
    </main>

    <footer>
        <p>&copy; 2025 {% block footer_name %}{% endblock %}</p>
    </footer>
</body>
</html>
```

Now create `templates/profile.html` that extends the base:

```
{% extends "base.html" %}

{% block title %}{{ profile.name }} - Profile{% endblock %}

{% block header_title %}{{ profile.name }}{% endblock %}

{% block content %}
<section id="about">
    <h2>About Me</h2>
    {% for paragraph in profile.about %}
    <p>{{ paragraph }}</p>
    {% endfor %}
</section>

<section id="skills">
    <h2>Skills</h2>
    <ul>
        {% for skill in profile.skills %}
        <li>{{ skill }}</li>
        {% endfor %}
    </ul>
</section>

<section id="contact">
    <h2>Contact</h2>
    {% if profile.email %}
    <p>Email: <a href="mailto:{{ profile.email }}">{{ profile.email }}</a></p>
    {% else %}
    <p>No contact information available.</p>
    {% endif %}
</section>
{% endblock %}

{% block footer_name %}{{ profile.name }}{% endblock %}
```

Look at how much cleaner this is. The loops and conditionals are right there in the HTML, easy to read. No string concatenation, no f-strings, no escaping issues.

Now update `main.py`:

```
from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates
from fastapi.staticfiles import StaticFiles

app = FastAPI()
templates = Jinja2Templates(directory="templates")
app.mount("/static", StaticFiles(directory="static"), name="static")
```

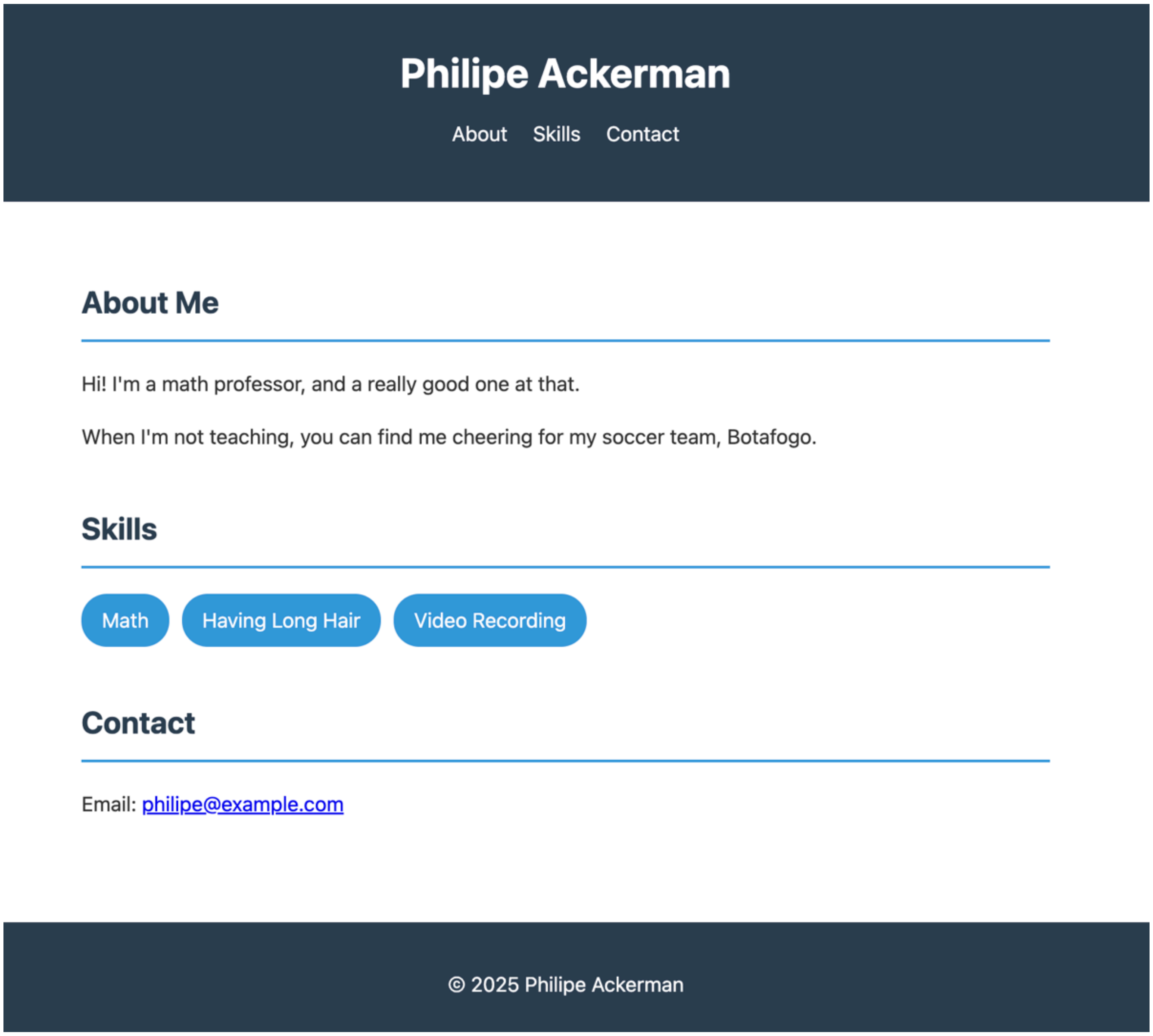
Add the profile data:

```
# main.py
# ...setup above

profile = {
    "name": "Philipe Ackerman",
    "about": [
        "Hi! I'm a math professor, and a really good one at that.",
        "When I'm not teaching, you can find me cheering for my soccer team, Botafogo.",
    ],
    "skills": ["Math", "Having Long Hair", "Video Recording"],
    "email": "philipe@example.com",
}
```

And the route:

```
# main.py
# ...profile data above

@app.get("/")
def home(request: Request):
    return templates.TemplateResponse(
        request=request,
        name="profile.html",
        context={"profile": profile}
    )
```

Run the server and visit `http://localhost:8000`. You should see the same profile page as before:



Philipe Ackerman

About Skills Contact

About Me

Hi! I'm a math professor, and a really good one at that.

When I'm not teaching, you can find me cheering for my soccer team, Botafogo.

Skills

Math Having Long Hair Video Recording

Contact

Email: philipe@example.com

© 2025 Philipe Ackerman

Figure 5.3: The profile page using templates

The page looks identical, but now the code is organized properly. The HTML is in HTML files. The Python is in Python files. Each does what it's good at.

Try adding a second profile. Create a route that takes a username:

```
# main.py
# ...existing code above

profiles = {
    "philipe": {
        "name": "Philipe Ackerman",
        "about": [
            "Hi! I'm a math professor, and a really good one at that.",
            "When I'm not teaching, you can find me cheering for my soccer team, Botafogo."
        ],
        "skills": ["Math", "Having Long Hair", "Video Recording"],
        "email": "philipe@example.com",
    },
    "matheus": {
        "name": "Matheus",
        "about": [
            "Applied math student at UFRJ interested in product and strategy.",
            "Part of the management team at the Applied Computing League.",
        ],
        "skills": ["Product Management", "UX Research", "Strategy", "Analytics"],
        "email": "matheus@example.com",
    },
}
```

Now add a route that uses the username to look up the profile:

```
# main.py
# ...profiles dict above

@app.get("/profile/{username}")
def show_profile(request: Request, username: str):
    profile = profiles.get(username)
    if not profile:
        return templates.TemplateResponse(
            request=request,
            name="not_found.html",
            context={"username": username},
            status_code=404
        )
    return templates.TemplateResponse(
        request=request,
        name="profile.html",
        context={"profile": profile}
    )
```

Create `templates/not_found.html`:

```
{% extends "base.html" %}

{% block title %}Not Found{% endblock %}

{% block header_title %}Not Found{% endblock %}

{% block content %}
<section>
    <h2>Profile Not Found</h2>
    <p>No profile exists for "{{ username }}".</p>
    <p><a href="/">Go back home</a></p>
</section>
{% endblock %}

{% block footer_name %}My Site{% endblock %}
```

Now visit `http://localhost:8000/profile/philipe` and `http://localhost:8000/profile/`
Same template, different data, different pages. Visit `http://localhost:8000/profile/nobody` and you get a proper 404 page.

This is the power of templates. One HTML file serves unlimited profiles. Add a hundred users to the database and the template handles them all.

5.7 Testing with Interactive Documentation

Remember the `/docs` page from Chapter 4? Now that we have a more interesting endpoint, let's actually use it.

Run your server and visit `http://localhost:8000/docs`. You'll see your endpoints listed. Click on the `/profile/{username}` endpoint to expand it. You'll see the parameter it expects: `username`, a required string. FastAPI figured this out from your code.

Click "Try it out" to enable the input field:

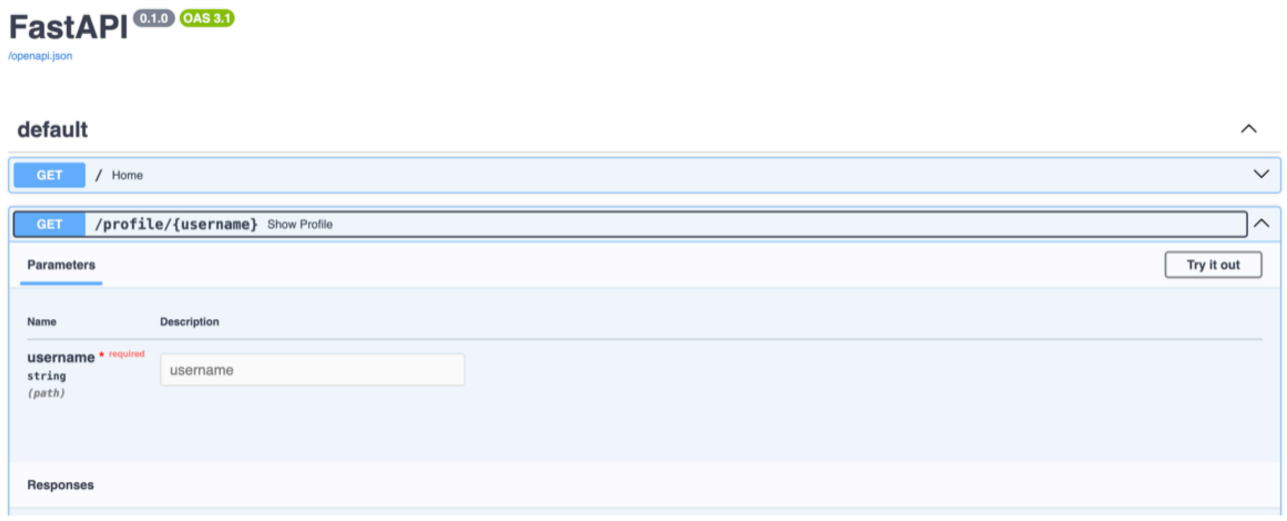
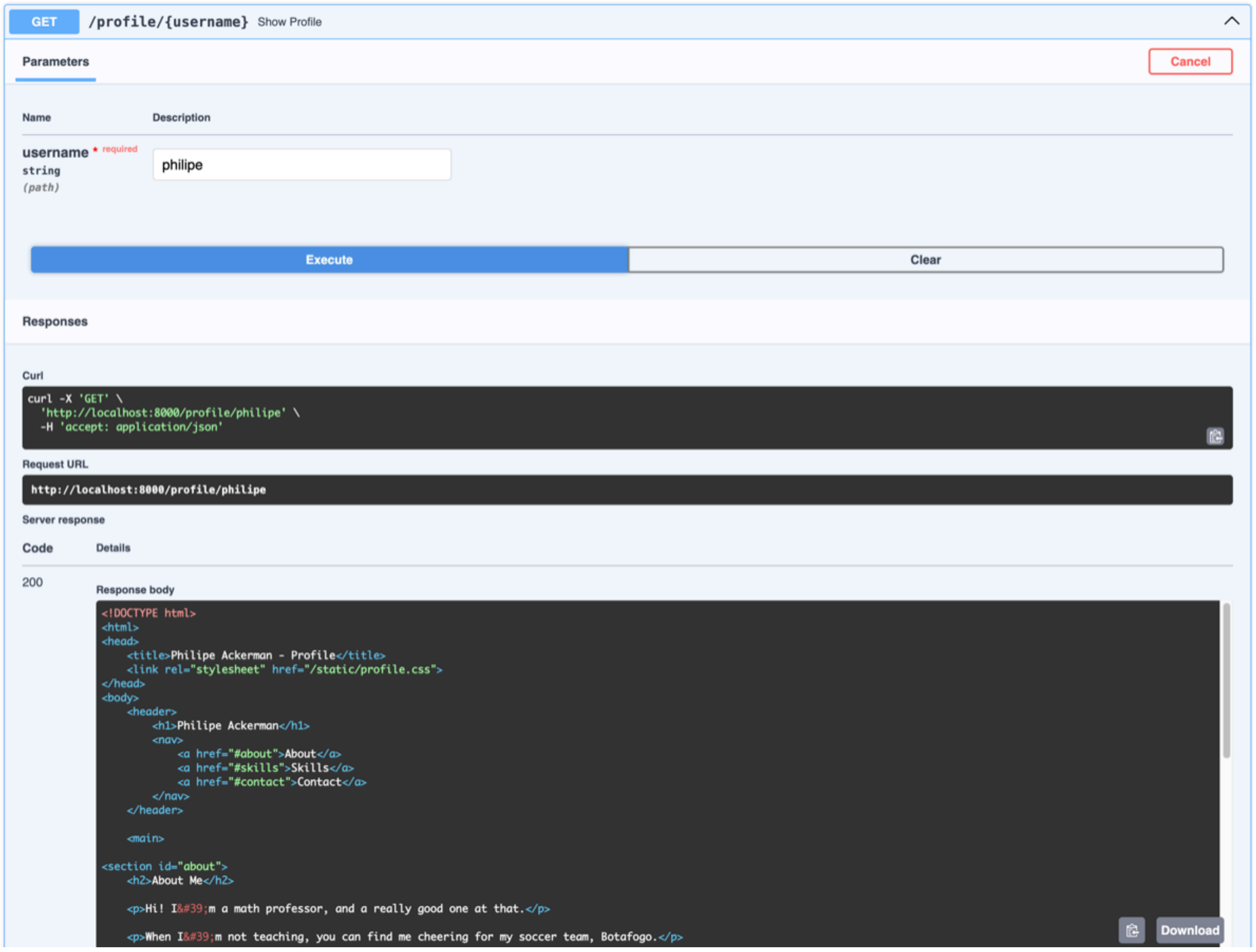


Figure 5.4: The endpoint expanded with input field ready

Type a username like `philipe` and click "Execute". The docs page makes a real request to your server and shows you the result:



```
curl -X 'GET' \
  'http://localhost:8000/profile/philipe' \
  -H 'accept: application/json'
```

```
http://localhost:8000/profile/philipe
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Philipe Ackerman - Profile</title>
    <link rel="stylesheet" href="/static/profile.css">
</head>
<body>
    <header>
        <h1>Philipe Ackerman</h1>
        <nav>
            <a href="#about">About</a>
            <a href="#skills">Skills</a>
            <a href="#contact">Contact</a>
        </nav>
    </header>

    <main>

<section id="about">
    <h2>About Me</h2>

    <p>Hi! I&#39;m a math professor, and a really good one at that.</p>

    <p>When I&#39;m not teaching, you can find me cheering for my soccer team, Botafogo.</p>
```

Figure 5.5: The response after executing the request

The response shows the actual URL it requested, the curl command to make the same request from a terminal, the status code (200 for success), and the response body containing the HTML your template generated.

Try a username that doesn't exist, like `nobody`. You'll see a 404 response with your not_found.html template.

This is useful for debugging. When something isn't working, you can test the endpoint directly without opening a browser tab, typing the URL, and refreshing. You see exactly what your server returns, including headers.

The documentation is generated from your code. The endpoint names come from your function names, the parameter types come from your type hints. If you add docstrings to your functions, they appear in the docs too:

```python
# main.py — update the existing show_profile function

@app.get("/profile/{username}")
def show_profile(request: Request, username: str):
    """
    Display a user's profile page.

    - **username**: The unique username to look up
    """
    profile = profiles.get(username)
    # ...rest of the function
```

Refresh `/docs` and you'll see the description appear. This matters when you're building APIs that other developers will use. They can read your documentation without digging through your code.

5.8 Chapter Summary

- Templates separate HTML from Python code
- Jinja uses `{ }` for outputting values and `{% %}` for logic
- Pass data to templates through the context dictionary
- `{% for %}` loops over lists; `{% if %}` handles conditionals
- Template inheritance (`{% extends %}` and `{% block %}`) eliminates repetition
- One template can serve unlimited variations of a page

In the next chapter, we'll handle forms: how to receive data from users and do something with it.

5.9 Exercises

1. Add a "projects" section to the profile page. Store a list of project dictionaries (each with name and description) in the profile data and display them using a loop.
2. Create an "index" page that lists all profiles with links to each one. You'll need a new template and a new route.
3. Add a conditional to the profile template that shows "No skills listed" if the skills list is empty. Test it by creating a profile with an empty skills list.
4. Create a second base template called `base_minimal.html` that has no header or footer. Make the 404 page extend this instead. This shows you can have multiple base templates for different page types.
5. Add a "theme" field to profiles (either "light" or "dark"). Use a conditional in the base template to add a CSS class to the body tag based on this field. You'll need to pass the profile to the base template somehow (hint: the profile is already in context).

6. Jinja has filters that transform values. Look up the `|title` and `|length` filters in the Jinja documentation. Use `|title` to capitalize names and `|length` to show how many skills someone has.

6 Chapter 6: Forms and User Input

In Chapter 2, we built HTML forms. In Chapter 5, we rendered pages with templates. But those forms didn't go anywhere. You could fill them out and click submit, and the browser would make a request, but nobody was listening on the other end. The form's `action` pointed to a URL that didn't exist on our server.

This chapter connects the two sides. You'll write server code that receives form data, validates it, and does something useful with it.

6.1 How Form Data Travels

When you click a submit button, the browser collects every input field in the form (specifically, every field with a `name` attribute), packages the data, and sends it to the server as an HTTP request. The form's `action` attribute determines the URL, and the `method` attribute determines whether it's a GET or POST request.

Consider this form:

```html
<form action="/search" method="GET">
    <input type="text" name="q" value="python">
    <input type="number" name="limit" value="10">
    <button type="submit">Search</button>
</form>
```

When submitted, the browser makes this request:

```
GET /search?q=python&limit=10 HTTP/1.1
Host: localhost:8000
```

The form data ends up in the query string, just like the URL parameters we saw in Chapters 1 and 4. Each input's `name` becomes a key, its value becomes the value.

Now consider a POST form:

```html
<form action="/contact" method="POST">
    <input type="text" name="name" value="Matheus">
    <input type="email" name="email" value="matheus@example.com">
    <textarea name="message">Hello there</textarea>
    <button type="submit">Send</button>
```

```
</form>
```

The request looks different:

```
POST /contact HTTP/1.1
Host: localhost:8000
Content-Type: application/x-www-form-urlencoded
Content-Length: 53

name=Matheus&email=matheus%40example.com&message=Hello+there
```

The data isn't in the URL. It's in the request body, after the headers. The format looks similar (key=value pairs separated by &), but notice the encoding: the `@` became `%40` and the space became +. This is URL encoding, and the browser handles it automatically. Special characters get encoded for safe transmission.

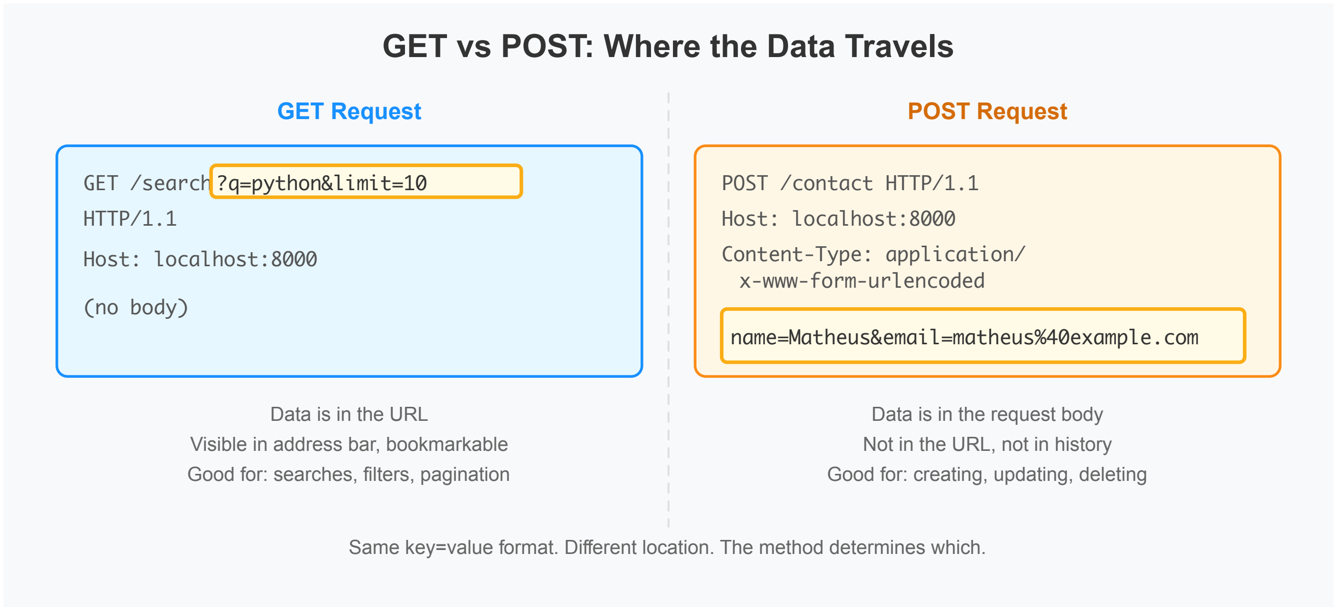

GET vs POST: Where the Data Travels
GET Request
POST Request
GET /search?q=python&limit=10
HTTP/1.1
Host: localhost:8000
(no body)
POST /contact HTTP/1.1
Host: localhost:8000
Content-Type: application/
x-www-form-urlencoded
name=Matheus&email=matheus%40example.com
Data is in the URL
Visible in address bar, bookmarkable
Good for: searches, filters, pagination
Data is in the request body
Not in the URL, not in history
Good for: creating, updating, deleting
Same key=value format. Different location. The method determines which.

Figure 6.1: GET vs POST: where the data travels

The `Content-Type: application/x-www-form-urlencoded` header tells the server how to parse the body. This is the default encoding for HTML forms. The data always travels as text, regardless of what type of input field it came from. A number input sends the string `"10"`, not the integer 10. Your server code needs to handle that conversion.

So GET and POST both send key-value pairs, but they put them in different places. The question is when to use which.

6.2 GET vs POST for Forms

Both methods send data to the server, but they're meant for different situations.

GET puts data in the URL. This means the data is visible in the address bar, saved in browser history, and bookmarkable. When you search for something on Google, the search term is in the URL: `google.com/search?q=python`. You can copy that URL, send it to someone, and they'll see the same results. GET requests are for reading data, not changing it. Searches, filters, pagination: all GET.

POST puts data in the request body. It's not visible in the address bar, not saved in history, and not bookmarkable. POST requests are for actions that change something: creating an account, sending a message, updating a profile, placing an order. If the operation writes data or has side effects, use POST.

The practical test: if a user refreshes the page after submitting, should the action happen again? For a search, yes (GET). For placing an order, absolutely not (POST). Browsers reinforce this by showing a warning when you try to refresh after a POST submission ("Are you sure you want to resubmit the form?").

There's also a size consideration. URLs have practical length limits (around 2000 characters in most browsers). A search query fits fine. A blog post doesn't. POST has no such limit because the data is in the body.

Most application features (creating things, logging in, sending messages) use POST forms, so that's what we'll focus on. GET forms for search are simpler because FastAPI already handles query parameters, as we saw in Chapter 4. But how does FastAPI know the difference between a query parameter and form data in a POST body?

6.3 Receiving Form Data in FastAPI

The answer is a new import: `Form`. Let's build a minimal example to see how it works. Create a new project for this chapter:

```
mkdir chapter6
cd chapter6
uv init
uv add "fastapi[standard]"
```

Set up the folder structure:

```
chapter6/
├── .venv/
├── main.py
├── pyproject.toml
└── templates/
    └── hello.html
```

In `main.py`:

```
# main.py
from fastapi import FastAPI, Request, Form
from fastapi.templating import Jinja2Templates

app = FastAPI()
templates = Jinja2Templates(directory="templates")
```

The new import here is `Form`. This is how FastAPI knows a parameter comes from form data rather than a query string or path.

Now create a page with a simple form and an endpoint to handle it:

```
# main.py
# ...imports and setup above

@app.get("/")
def show_form(request: Request):
    return templates.TemplateResponse(
        request=request,
        name="hello.html",
        context={"greeting": None}
    )


@app.post("/")
def handle_form(request: Request, name: str = Form()):
    return templates.TemplateResponse(
        request=request,
        name="hello.html",
        context={"greeting": f"Hello, {name}!"}
    )
```

Two relevant things to notice. GET and POST to the same URL (/) trigger different functions. The browser visiting the page is a GET. Submitting the form is a POST.

The parameter `name: str = Form()` tells FastAPI to look for a value called `name` in the form data of the POST request body. Without `Form()`, FastAPI would look for a query parameter. The `Form()` default is what tells it to parse the request body instead.

The template in `templates/hello.html`:

```
<!DOCTYPE html>
<html>
<head><title>Hello</title></head>
<body>
    {% if greeting %}
    <p>{{ greeting }}</p>
    {% endif %}

    <form action="/" method="POST">
        <label for="name">Your name</label>
        <input type="text" id="name" name="name" required>
        <button type="submit">Say Hello</button>
    </form>
</body>
</html>
```

Run it with `uv run fastapi dev main.py`, visit `http://localhost:8000`, type your name, and submit. The page reloads with a greeting. Open the Network tab and watch the POST request. The form data is in the request body, and the `name` attribute on the input matches the `Form()` parameter in your Python function.

The full cycle of what just happened is this:

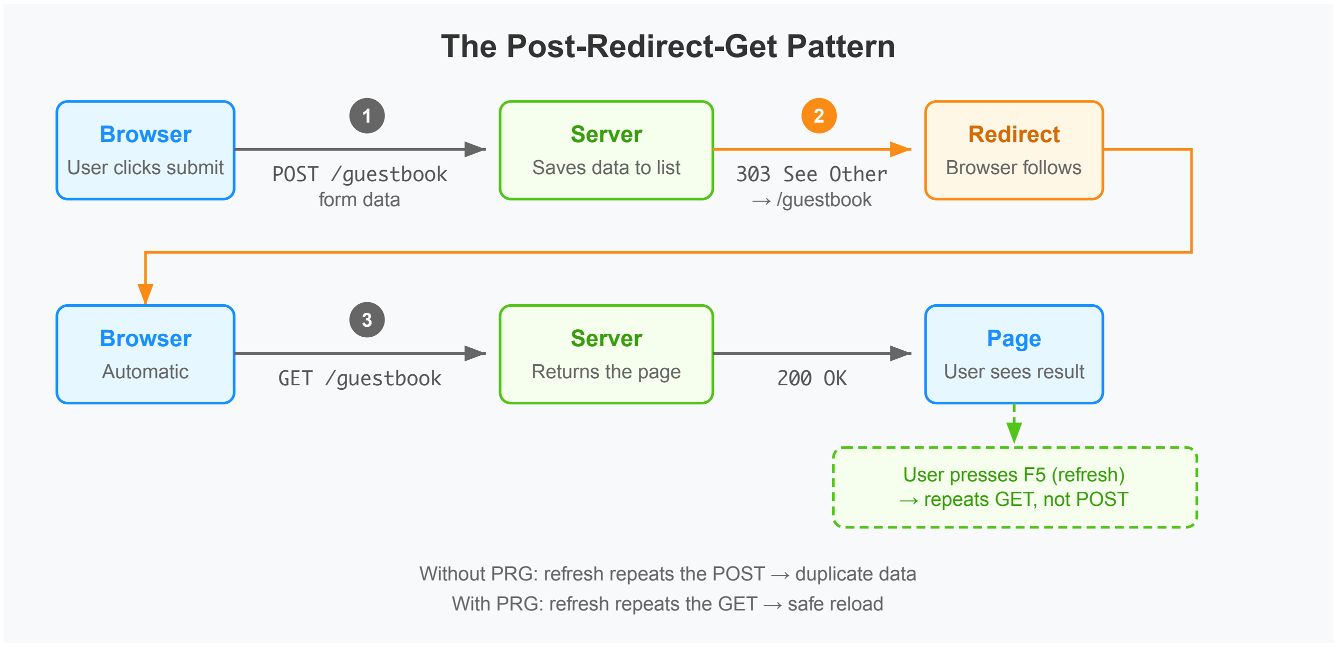


Figure 6.2: The form submission cycle

The browser sends the form data in a POST request. FastAPI extracts the `name` parameter from the body, passes it to your function, and your function returns HTML through the template. This is the same request-response cycle from Chapter 1, but now with data flowing from user to server.

It works, but there's a problem. Type a single space, hit submit, and you get "Hello, !" with an awkward gap. The server accepts anything.

6.4 Validation

In a real application, bad input causes real problems: empty rows in your database, broken layouts, security vulnerabilities. Validation means checking that input meets your requirements before you do anything with it. There are two places to validate: in the browser (client-side) and on the server (server-side). You need both, in the client for a good user experience and on the server for security reasons.

Client-side validation gives users immediate feedback. The `required` attribute prevents empty submissions. You can also use `minlength`, `maxlength`, and `pattern`:

```
<input type="text" name="name" required minlength="2" maxlength="100">
```

But client-side validation is easily bypassed. Anyone can open developer tools, remove the `required` attribute, and submit. Or send a POST request directly with curl, skipping the browser entirely. Server-side validation is the one that actually protects your application.

Let's add validation to our endpoint:

```
# main.py
# replace the existing handle_form function

@app.post("/")
def handle_form(request: Request, name: str = Form()):
    name = name.strip()

    if not name:
        return templates.TemplateResponse(
            request=request,
            name="hello.html",
            context={"greeting": None, "error": "Name is required."}
        )

    return templates.TemplateResponse(
        request=request,
        name="hello.html",
        context={"greeting": f"Hello, {name}!"}
    )
```

`strip()` removes whitespace from both ends. A name that's just spaces becomes an empty string, which fails the check. When validation fails, we re-render the form with an error message instead of a greeting.

Update the template to show errors:

```
<!DOCTYPE html>
<html>
<head><title>Hello</title></head>
<body>
    {% if greeting %}
    <p>{{ greeting }}</p>
    {% endif %}

    {% if error %}
    <p style="color: red;">{{ error }}</p>
    {% endif %}
    <form action="/" method="POST">
        <label for="name">Your name</label>
        <input type="text" id="name" name="name" required>
        <button type="submit">Say Hello</button>
    </form>
</body>
</html>
```

Try it: use dev tools to remove the `required` attribute, submit an empty form, and you'll see the error. The browser validation is a convenience for users, the server validation is the actual safety net.

For forms with more fields, you'd collect errors into a list and pass the user's input back so they don't have to retype everything:

```
errors = []
if not name:
    errors.append("Name is required.")
if not email:
    errors.append("Email is required.")

if errors:
    return templates.TemplateResponse(
        request=request,
        name="form.html",
        context={"errors": errors, "form_name": name, "form_email": email}
    )
```

Then in the template, `value="{{ form_name or '' }}"` preserves what the user typed.

This works fine for a form with two fields. But validation logic gets repetitive fast: strip this, check if it's empty, check if it's too long, check if it's a valid email. For each field. Every time. Pydantic gives you a cleaner way.

6.5 Validation with Pydantic

Pydantic is a library for defining data shapes and validation rules in one place. FastAPI already uses it under the hood (it's how type hints like `user_id: int` get validated automatically in path parameters). Now we'll use it directly.

A Pydantic model is a class that describes what valid data looks like:

```
from pydantic import BaseModel, Field

class HelloForm(BaseModel):
    name: str = Field(min_length=1, max_length=100)
```

This says: `name` must be a string, at least 1 character, at most 100. If you try to create a `HelloForm` with invalid data, Pydantic raises a `ValidationError`:

```
from pydantic import ValidationError

try:
    data = HelloForm(name="")
except ValidationError as e:
    print(e.errors())
    # [{'type': 'string_too_short',
    #   'msg': 'String should have at least 1 character',
    #   ...}]
```

The error messages are generated automatically from your constraints. You don't write them yourself (though you can customize them if you want).

Here's how this fits into a FastAPI endpoint:

```
# main.py
# replace the existing handle_form function

from pydantic import BaseModel, Field, ValidationError

class HelloForm(BaseModel):
    name: str = Field(min_length=1, max_length=100)


@app.post("/")
def handle_form(request: Request, name: str = Form()):
    try:
        data = HelloForm(name=name.strip())
    except ValidationError as e:
        errors = [err["msg"] for err in e.errors()]
        return templates.TemplateResponse(
            request=request,
            name="hello.html",
            context={"greeting": None, "errors": errors}
        )

    return templates.TemplateResponse(
        request=request,
        name="hello.html",
        context={"greeting": f"Hello, {data.name}!"}
    )
```

We still use `Form()` to extract the raw string from the request body. Then we pass it to the Pydantic model, which validates it. If validation fails, we catch the error and extract the messages. If it succeeds, we use `data.name` (the validated, clean value).

The pattern is: `Form()` gets the data out of the request, Pydantic checks whether it's valid.

For a single field, this might seem like more code than the manual `if not name` check. The payoff comes with more fields and more rules. Here's a model for a contact form:

```
class ContactForm(BaseModel):
    name: str = Field(min_length=1, max_length=100)
    email: str = Field(min_length=1, max_length=200)
    message: str = Field(min_length=1, max_length=2000)
```

Three fields, six constraints, zero if statements. You can add more rules with `pattern` for regex matching, or write custom validators with `@field_validator` when the built-in constraints aren't enough. The Pydantic documentation covers these in detail.

One more thing about security: what if someone submits `<script>alert('hi')</script>` as their name? Jinja auto-escapes HTML in template variables by default, so the script tag gets rendered as harmless text, not executed. This is one of the reasons we use a template engine instead of building HTML strings manually.

Our form now validates data properly, but there's still a usability problem. Submit a name, see the greeting, then press F5 to refresh. The browser shows a warning about resubmitting form data.

6.6 Post-Redirect-Get

This happens because refreshing repeats the last request. If the last request was a POST, the browser sends the same POST again. For our hello form this is harmless. But imagine a form that places an order or sends an email. Refreshing would place the order twice or send duplicate messages.

The fix is a pattern called **Post-Redirect-Get** (PRG). Instead of returning HTML directly from the POST handler, you redirect the user to a GET page:

1. User submits form (POST)
2. Server processes the data
3. Server responds with a redirect (HTTP 303) to a GET URL
4. Browser follows the redirect and loads the page via GET
5. User sees the result

Now if the user refreshes, they're refreshing a GET request. No resubmission.

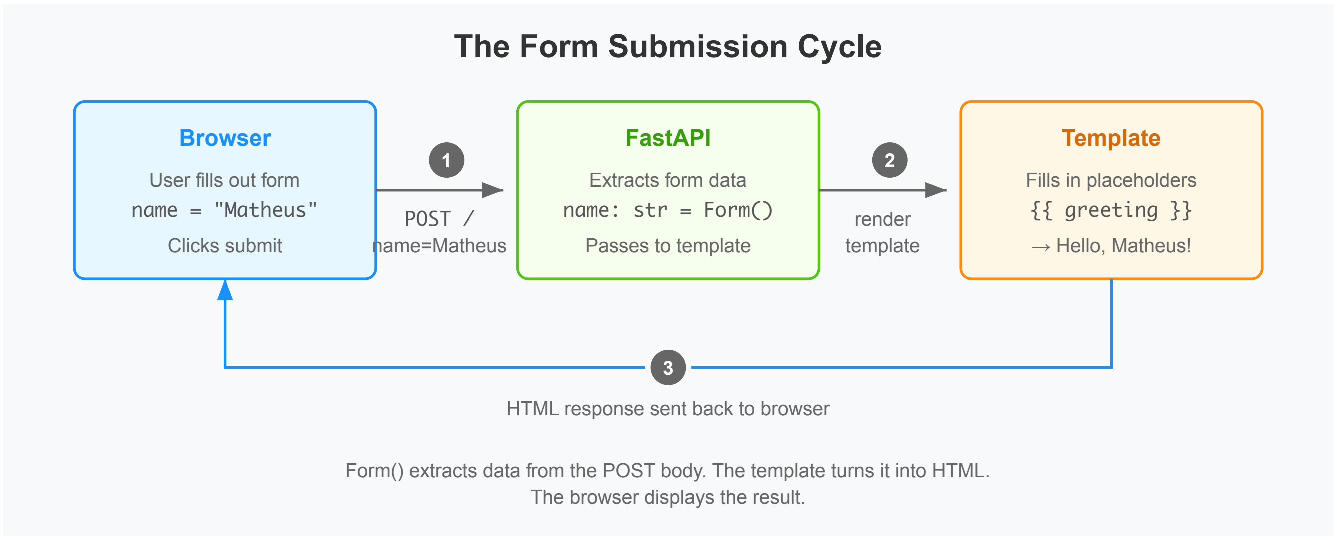


Figure 6.3: The Post-Redirect-Get pattern

The browser handles the redirect automatically. The user doesn't notice it happened. Let's update our example:

```
# main.py
# updated imports
from fastapi import FastAPI, Request, Form
from fastapi.responses import RedirectResponse
from fastapi.templating import Jinja2Templates
```

Then replace the existing handle_form function:

```
# main.py

@app.post("/")
def handle_form(request: Request, name: str = Form()):
    try:
        data = HelloForm(name=name.strip())
    except ValidationError as e:
        errors = [err["msg"] for err in e.errors()]
        return templates.TemplateResponse(
            request=request,
            name="hello.html",
            context={"greeting": None, "errors": errors}
        )

    # In a real app, you'd save the data here
    return RedirectResponse(url=f"/?greeting={data.name}", status_code=303)
```

The `status_code=303` tells the browser to follow the redirect with a GET request. We're passing the name through the URL so the GET handler can display it:

```python
# main.py
# replace the existing show_form function

@app.get("/")
def show_form(request: Request, greeting: str | None = None):
    context = {"greeting": f"Hello, {greeting}!" if greeting else None}
    return templates.TemplateResponse(
        request=request,
        name="hello.html",
        context=context
    )
```

Submit a name, then refresh. No warning. No resubmission.

Passing data through the URL query string works for this tiny example, but it's awkward for real applications. When we build the guestbook in the hands-on, the data goes into a list (and later, a database). The redirect just sends the user to the page that displays the stored data. The pattern is: POST saves the data, redirect goes to the page, GET reads and displays it.

When validation fails, we still render the template directly (no redirect). The user needs to see the errors and their partially filled form. PRG only happens on success.

That covers all the pieces. Time to apply them to the profile site we've been building.

6.7 Hands-On: Adding a Guestbook to the Profile Site

Our profile site from Chapter 5 has profile pages, a base template, and static CSS. We're going to add a new page: a guestbook where visitors can leave messages. This means a new template, new routes, and new CSS, all integrated with the existing site.

Start from your Chapter 5 project. If you don't have it handy, grab it from the repository. Your folder structure should currently look like this:

```
chapter5/
├── .venv/
├── main.py
├── pyproject.toml
├── static/
│   └── profile.css
└── templates/
    ├── base.html
    ├── not_found.html
    └── profile.html
```

Copy it into a new folder for this chapter:

```
cp -r chapter5 chapter6
cd chapter6
```

We need a few new files. Add a guestbook template and rename the CSS file (it's no longer just for profiles):

```
chapter6/
├── .venv/
├── main.py
├── pyproject.toml
├── static/
│   └── style.css
└── templates/
    ├── base.html
    ├── guestbook.html
    ├── not_found.html
    └── profile.html
```

Rename `profile.css` to `style.css` and update the `<link>` in `base.html` to match. Then add a guestbook link to the navigation in `base.html`:

```
<nav>
    <a href="/">Home</a>
    <a href="/guestbook">Guestbook</a>
    <a href="/about">About</a>
</nav>
```

Now update `main.py`. You'll need the new imports we learned about in this chapter:

```
# main.py
# add to the existing imports
from datetime import datetime
from pydantic import BaseModel, Field, ValidationError
from fastapi import FastAPI, Request, Form
from fastapi.responses import RedirectResponse
```

Add a Pydantic model for the guestbook form and storage for messages:

```
# main.py
# ...imports and setup above

class GuestbookEntry(BaseModel):
    name: str = Field(min_length=1, max_length=100)
    message: str = Field(min_length=1, max_length=500)

messages = []
```

Now you need to write three things with these requirements:

A GET endpoint at `/guestbook` that renders `guestbook.html` with the messages list in the context.

A POST endpoint at `/guestbook` that:

- Receives `name` and `message` via `Form()`
- Strips whitespace and validates with the `GuestbookEntry` model
- On failure: re-renders the template with errors and the user's input preserved
- On success: appends a dictionary (with `name`, `message`, and a formatted `time`) to the messages list and redirects to `/guestbook`

For the timestamp, `datetime.now().strftime("%B %d, %Y at %I:%M %p")` gives you something like "January 15, 2025 at 02:30 PM". Store each message as a dictionary with `name`, `message`, and `time` keys.

A POST endpoint at `/guestbook/delete/{index}` that removes a message by index and redirects to `/guestbook`. Include a bounds check so invalid indices don't crash the server.

Now create `templates/guestbook.html`. It should extend `base.html` and include these:

- An error display section (if errors exist in the context)
- A form with `action="/guestbook"` and `method="POST"` containing a text input for `name`, a textarea for `message`, and a submit button
- The inputs should preserve the user's data on validation failure (using `value="{{ form_name or '' }}"` for the input and `{ form_message or '' }` between the textarea tags)
- A messages section that loops through `messages` and displays each one's name, time, and message
- A delete button for each message (a small form that POSTs to `/guestbook/delete/{{ loop.index0 }}`)

Here's a skeleton to get you started:

```
{% extends "base.html" %}

{% block title %}Guestbook{% endblock %}

{% block content %}
<h1>Guestbook</h1>

<!-- TODO: show errors if they exist -->

<form action="/guestbook" method="POST">
    <!-- TODO: name input and message textarea -->
    <!-- TODO: preserve user input on validation failure -->
    <button type="submit">Sign Guestbook</button>
</form>

{% if messages %}
<h2>Messages</h2>
```

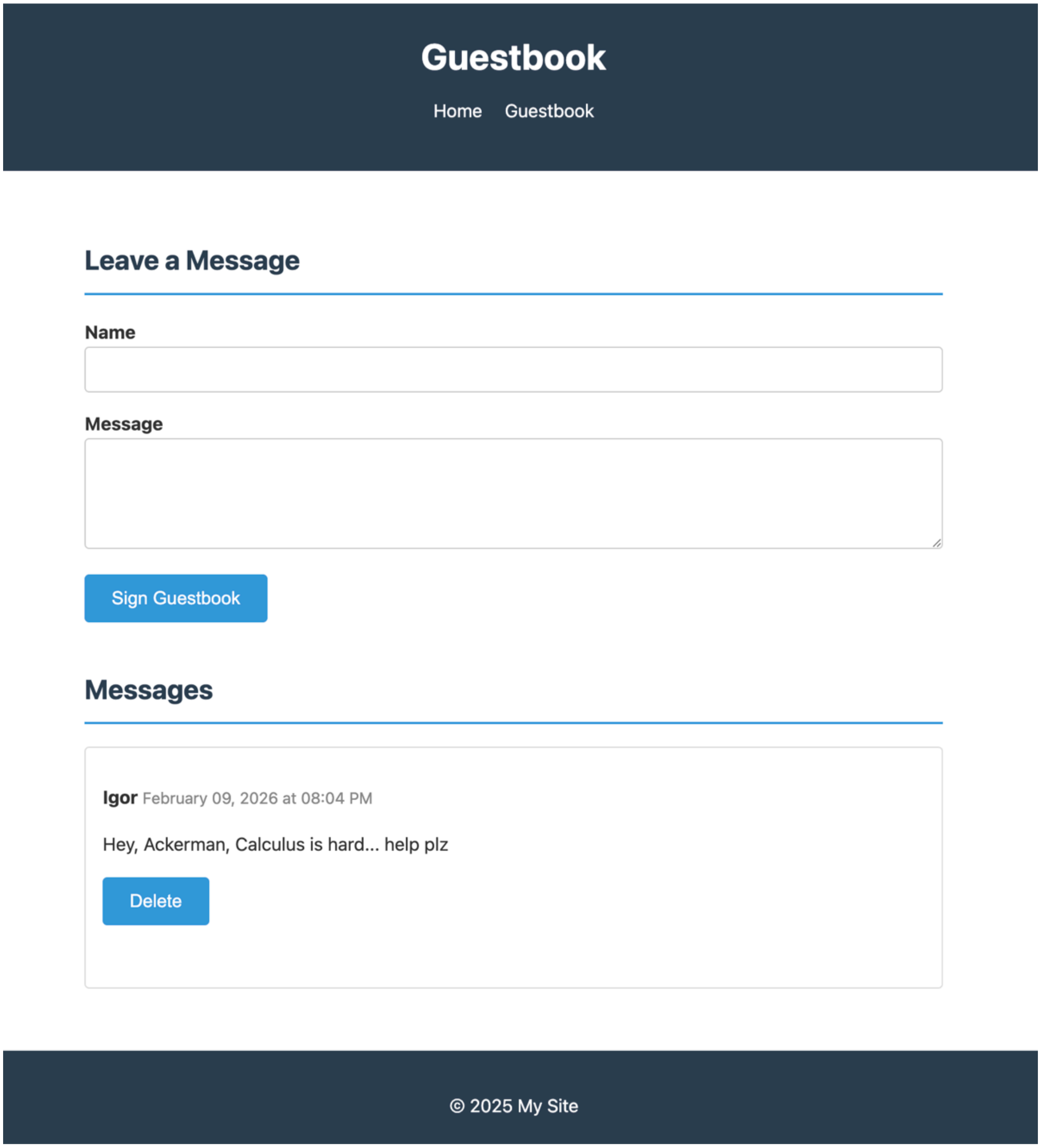
```
{% for msg in messages %}
<div class="message">
    <!-- TODO: display msg.name, msg.time, msg.message -->
    <!-- TODO: delete button (its own small form) -->
</div>
{% endfor %}
{% endif %}
{% endblock %}
```

For the CSS, add guestbook styles to `static/style.css`. You need styles for the form layout (stacked labels and inputs), form inputs and the submit button, message cards with the name and timestamp, an error display with a red border, and the delete button (subtle, not a full button). You've styled forms and cards before in Chapters 3 and 5. Apply the same principles.

Test your work:

- Submit a few messages. Do they appear with timestamps?
- Submit with empty fields (use dev tools to bypass the `required` attribute). Do you see Pydantic's error messages? Is your input preserved?
- Submit a valid message, then refresh. Do you get the resubmission warning? (You shouldn't, if PRG is working.)
- Delete a message. Does it disappear?
- Check the Network tab. Can you see the POST □ 303 □ GET sequence?
- Navigate between profile pages and the guestbook. Does the nav work?

When everything works, your guestbook should look something like this (your styling will vary):



Igor February 09, 2026 at 08:04 PM

Hey, Ackerman, Calculus is hard... help plz

Figure 6.4: The guestbook page integrated with the profile site

Stuck? Solution available

If you've given it an honest try and are truly stuck, a completed version of this project is available at github.com/Applied-Computing-League/practical-web-development/en/code/chapter6. Try to solve it yourself first. You learn more from the struggle than from reading the answer.

6.8 Chapter Summary

- HTML forms send data to the server as key-value pairs via GET (query string) or POST (request body)
- Use GET for reading (searches, filters) and POST for actions that change data
- In FastAPI, `Form()` tells a parameter to read from the request body instead of the query string
- Pydantic models define validation rules in one place: `Field(min_length=1, max_length=100)`
- Always validate on the server, even if you also validate in the browser
- Post-Redirect-Get prevents duplicate submissions on refresh: process the POST, redirect to a GET page
- When validation fails, re-render the form with errors and the user's input preserved

6.9 Exercises

1. Add a "character count" display below the message textarea that updates as the user types. This requires a small amount of JavaScript: add a `<script>` tag at the bottom of your template that listens for the `input` event on the textarea and updates a `<span>` with the current length. You haven't learned JavaScript yet, but you can figure this out (or ask for a hint). This is client-side only; the server doesn't need to change.

2. The delete endpoint has no protection. Anyone who knows the URL can delete messages. Add a simple "admin password" check: the delete form should include a hidden input with a password, and the endpoint should only delete if the password matches a value you hardcode in your Python code. This isn't real security (we'll cover authentication later), but it demonstrates how hidden form fields work.

3. Add an optional "email" field to the guestbook. Update the Pydantic model to allow it (hint: use `str | None = None` with `Field(default=None, max_length=200)`). When provided, display it next to the name. Add a custom Pydantic validator using `@field_validator` that checks the email contains an `@` symbol if it's not None.

4. Right now messages are lost when the server restarts. Use Python's `json` module to save messages to a JSON file when a message is added or deleted, and load them when the server starts. The file should be created automatically if it doesn't exist.

5. Add pagination: only show the 10 most recent messages, with a “Show all” link that displays everything. You’ll need a query parameter (`?all=true`) and a conditional in the template. Think about whether the messages should be stored newest-first or oldest-first, and why.

6. Test your validation by bypassing the browser. Use the `/docs` endpoint to submit a POST request with empty fields, with fields that are too long, and with valid data. Watch how your server responds to each case. Then try submitting with curl from your terminal (the docs page shows you the curl command).

7 Chapter 7: Interactive Pages with HTMX

The guestbook from Chapter 6 works. You can submit messages, validate input, delete entries. But every single interaction reloads the entire page. Submit a message: the browser sends a POST, receives a redirect, requests the full page again, and re-renders everything from the `<html>` tag down. Delete a message: same thing. The nav, the header, the CSS, the guestbook form, every message that didn't change: all re-downloaded and re-rendered just to update one part of the page.

For a guestbook, this is fine. The page is small, the interactions are infrequent, and the reload is barely noticeable. But think about a search bar. You want results to appear as the user types. If every keystroke triggered a full page reload, the experience would be terrible: the page flashing white, the cursor losing focus, the scroll position resetting. Some interactions need to update part of the page without replacing all of it.

This chapter introduces HTMX, a library that lets you do exactly that without writing JavaScript.

7.1 Multi-Page vs Single-Page Applications

What we've been building is called a **multi-page application** (MPA). Every navigation and form submission is a full round trip: the browser sends a request, the server sends back a complete HTML page, and the browser replaces everything on screen. This is how the web worked for its first 15 years, and it's still how most websites work today.

In the mid-2000s, **single-page applications** (SPAs) emerged as an alternative. The browser loads one page, then JavaScript handles everything: intercepting link clicks, fetching JSON from the server, and redrawing parts of the page in place. Frameworks like React, Vue, and Angular work this way. SPAs can feel very fast because they only update what changed, but they come with real complexity. You're essentially building two applications: a server that serves data and a JavaScript frontend that turns that data into HTML. Routing, form handling, error states, loading indicators: all things the browser gives you for free in an MPA, all things you have to reimplement in JavaScript. We'll build one in a later chapter so you can see the trade-offs firsthand.

But there's a middle ground. What if you could keep your server-rendered HTML (the approach you already know) and just make it smarter about which parts of the page to replace? You wouldn't need a JSON API. You wouldn't need a JavaScript framework. You'd just add a few HTML attributes and let a small library handle the partial updates.

That's what HTMX does.

7.2 The HTMX Approach

HTMX is a JavaScript library, but you don't write any JavaScript to use it. You include it with a script tag and control it entirely through HTML attributes. The idea is simple: any HTML element can make an HTTP request, and the response (which is just HTML) gets swapped into the page.

You already know how to write endpoints that return HTML. In Chapter 5, your profile endpoint rendered a template and returned a full page. HTMX doesn't change how your server works. It changes what the server needs to return. Instead of always sending a complete page (with `<html>`, `<head>`, `<body>`, navigation, footer), you can send just the piece that changed: a search result list, a success message, a single updated row.

To add HTMX to a project, include the script in your base template's `<head>`:

```
<script src="https://unpkg.com/htmx.org@2.0.4"></script>
```

That's it. No build step, no npm, no configuration. The library is about 14KB, smaller than most images on a typical web page.

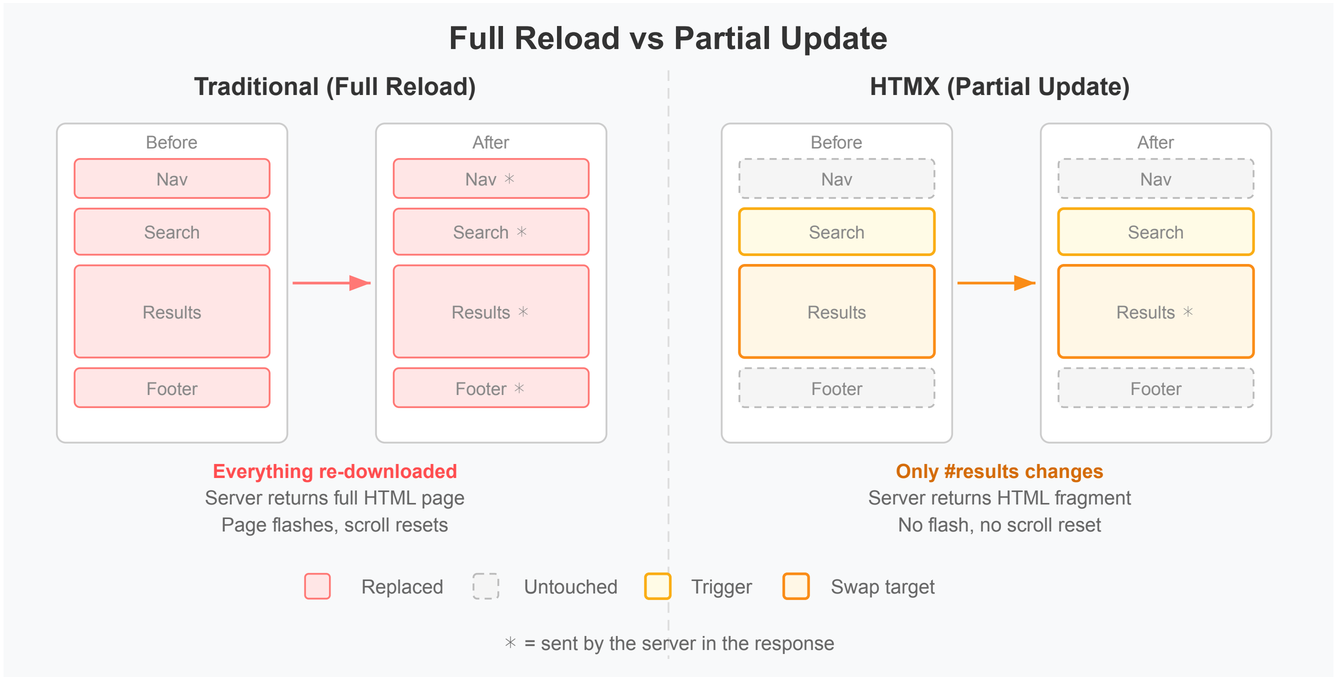

Full Reload vs Partial Update
Traditional (Full Reload)
HTMX (Partial Update)
Before
After
Nav
Search
Results
Footer
Nav ⁎
Search ⁎
Results ⁎
Footer ⁎
Everything re-downloaded
Server returns full HTML page
Page flashes, scroll resets
Only #results changes
Server returns HTML fragment
No flash, no scroll reset
Replaced
Untouched
Trigger
Swap target
⁎ = sent by the server in the response

Figure 7.1: Full page reload vs HTMX partial update

The difference is visible in the diagram. On the left, a traditional page reload: the server returns an entire HTML document, and the browser replaces everything, including the parts that didn't change. On the right, HTMX: the server returns just the piece that changed, and HTMX swaps it into the target element. The nav, the search input, the footer: all untouched.

To see why this is better, try building a search feature the way you already know how.

7.3 hx-get and hx-post

Say you want a page where users can search for profiles. With everything you've learned so far, you'd build it as a normal form:

```
<form action="/search" method="GET">
    <input type="text" name="q" placeholder="Search profiles...">
    <button type="submit">Search</button>
</form>

<div id="results">
    <!-- results rendered by the server -->
</div>
```

And an endpoint:

```
@app.get("/search")
def search(request: Request, q: str = ""):
    results = filter_profiles(q)
    return templates.TemplateResponse("search.html", {
        "request": request, "profiles": results, "query": q
    })
```

This works. The user types a name, clicks Search, and gets a page of results. But the entire page reloads. The browser fetches the full HTML document: `<html>`, `<head>`, navigation, footer, everything. The URL changes to `/search?q=lucas`. If the user wants to refine their search, they edit the input and click Search again. Another full reload.

Now try adding `hx-get` to the input instead:

```
<input type="text" name="q" placeholder="Search profiles..."
       hx-get="/search"
       hx-trigger="keyup changed delay:300ms"
       hx-target="#results">

<div id="results"></div>
```

And change the endpoint to return just the results, not a full page:

```
@app.get("/search")
def search(q: str = ""):
    results = filter_profiles(q)
    return HTMLResponse(render_results(results))
```

Type a few characters. The results appear below the input. No page reload, no flash, no lost cursor position. Open the Network tab and you'll see small requests going out as you type, each returning a fragment of HTML (not a full document). The nav, the header, the footer: none of them were re-sent or re-rendered.

`hx-get` tells HTMX: “when this element is triggered, make a GET request to this URL.” HTMX automatically includes the element’s `name` and `value` as query parameters, so the server receives `q=lucas` the same way it would from a form submission. The difference is what happens with the response. Instead of the browser loading a new page, HTMX takes the HTML fragment and swaps it into the current page.

`hx-post` works the same way but sends a POST request. This is useful for form submissions that change data:

```html
<form hx-post="/subscribe" hx-target="#subscribe-result">
    <input type="email" name="email" placeholder="you@example.com">
    <button type="submit">Subscribe</button>
</form>
<div id="subscribe-result"></div>
```

And the endpoint:

```python
@app.post("/subscribe")
def subscribe(email: str = Form()):
    return HTMLResponse("<p>Thanks! You're subscribed.</p>")
```

The endpoint uses `Form()` the same way you learned in Chapter 6. It returns HTML the same way you’ve been doing since Chapter 4. The only difference is that the response is a fragment (just a `<p>` tag) instead of a full page, and HTMX swaps it into `#subscribe-result` instead of reloading the page.

But there’s already something new in those examples that we glossed over: `hx-target` and `hx-trigger`. Without them, HTMX’s default behavior creates problems.

7.4 hx-target and hx-swap

By default, HTMX replaces the innerHTML of the element that made the request. Try this:

```html
<button hx-get="/greeting">Say hello</button>
```

With this endpoint:

```python
@app.get("/greeting")
def greeting():
    return HTMLResponse("<p>Hello from the server!</p>")
```

Click the button. The greeting appears, but it replaces the button’s text. The button now says “Hello from the server!” inside a `<p>` tag. Click it again and… nothing visible happens, because the `<p>` is already there. The button effectively destroyed itself.

This is rarely what you want. A search input shouldn't replace itself with results. A "load more" button shouldn't become the content it loaded. You need to separate the element that triggers the request from the element that displays the response.

`hx-target` takes a CSS selector and tells HTMX where to put the response:

```
<button hx-get="/greeting" hx-target="#output">Say hello</button>
<div id="output"></div>
```

Now clicking the button fetches `/greeting`, but the response goes into `#output`. The button stays visible. You can click it again.

This separation is what makes HTMX practical. A search input puts results in a list below it. A delete button removes its parent row. A form appends new items to a container elsewhere on the page.

Once you control where the response goes, the next question is how it gets inserted. The default (`innerHTML`) replaces the target's contents. That works for search results, where you want the new results to replace the old ones. But what about a chat or a message list, where you want to add to what's already there?

`hx-swap` controls the insertion method. `innerHTML` is the default: replace the target's contents. `beforeend` appends to the end:

```
<form hx-post="/messages" hx-target="#message-list" hx-swap="beforeend">
    <input name="message">
    <button type="submit">Send</button>
</form>

<div id="message-list">
    <!-- existing messages stay, new ones appear at the bottom -->
</div>
```

Each submission adds a new message without removing the existing ones. `afterbegin` does the same but prepends (newest first).

`outerHTML` replaces the target element itself, not just its contents. This is useful when the response is a replacement for the entire component:

```
<div id="status" hx-get="/check-status" hx-swap="outerHTML">
    Checking...
</div>
```

The server returns a new `<div id="status">` with the updated content, and the old one is swapped out entirely. You'll use `outerHTML` in Exercise 5 for a "load more" button that replaces itself with new content and a fresh button.

Those three swap modes cover most situations. HTMX has a few more (`beforebegin`, `afterend`, `delete`, `none`), but you can look those up when you need them.

One thing in the search example still needs explaining. We used `hx-trigger="keyup changed delay:300ms"` on the input, but we haven't discussed what that means.

By default, buttons trigger on `click` and inputs trigger on `change` (which fires when the input loses focus, not while you're typing). For live search, you want the request to fire as the user types. `hx-trigger` lets you customize this:

```
<input type="text" name="q"
       hx-get="/search"
       hx-target="#results"
       hx-trigger="keyup changed delay:300ms">
```

`keyup` fires every time a key is released. `changed` skips requests when the value hasn't actually changed (like pressing arrow keys or Shift). `delay:300ms` waits 300 milliseconds of inactivity before sending the request, so rapid typing produces one request at the end rather than one per keystroke. Without the delay, typing "lucas" would fire five requests (`l`, `lu`, `luc`, `luca`, `lucas`). With it, the user gets a brief pause to finish typing and then one request goes out.

This combination of `hx-get`, `hx-target`, `hx-trigger`, and `hx-swap` covers most real-world HTMX use cases. The HTMX documentation at htmx.org covers additional features (pushing URLs to the address bar, confirmation dialogs, server-sent events) when you need them.

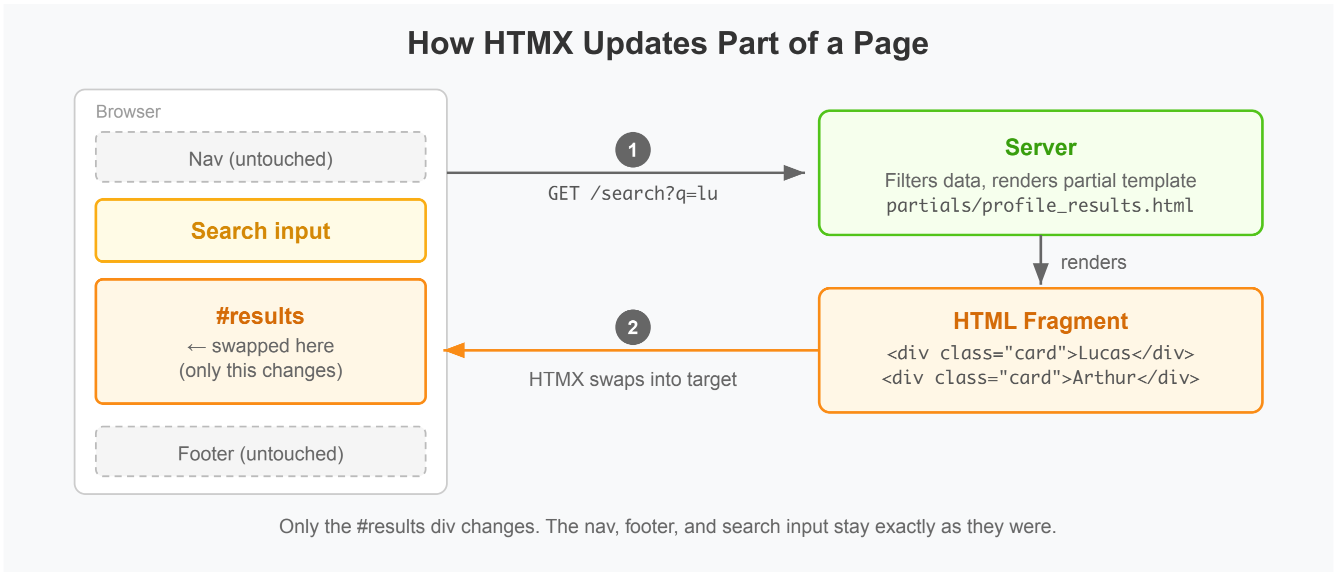

How HTMX Updates Part of a Page
Browser
Nav (untouched)
Search input
#results
← swapped here
(only this changes)
Footer (untouched)
1
GET /search?q=lu
Server
Filters data, renders partial template
partials/profile_results.html
renders
HTML Fragment
<div class="card">Lucas</div>
<div class="card">Arthur</div>
2
HTMX swaps into target
Only the #results div changes. The nav, footer, and search input stay exactly as they were.

Figure 7.2: How HTMX updates part of a page

The diagram shows the flow when you type in a search box with HTMX. The browser sends a request for just the results, the server renders a fragment (not a full page), and HTMX swaps it into the target div. The rest of the page is untouched.

There's one more piece of user experience to address. When you type in the search box, something happens on the server, and then results appear. But what does the user see during that gap?

7.5 Loading States

On a fast local server, the gap between request and response is almost invisible. But over a real network, with a database query behind the endpoint, there could be a noticeable pause. During that pause, the user has no feedback that anything is happening. They typed, nothing changed, and they don't know if the app is working or broken.

HTMX handles this with CSS. When a request is in flight, HTMX adds the class `htmx-request` to the triggering element. When the response arrives, it removes the class. You can hook into this with a simple CSS rule to show a loading indicator:

```
.search-spinner {
    opacity: 0;
    transition: opacity 200ms;
    position: absolute;
}
.search-spinner.htmx-request {
    opacity: 1;
}
```

And mentioning it in our html:

```
<div class="search-box" style="position: relative;">
    <input type="text" name="q"
           hx-get="/search"
           hx-target="#results"
           hx-trigger="keyup changed delay:300ms"
           hx-indicator=".search-spinner">
    <span class="search-spinner">Searching...</span>
</div>
```

The `hx-indicator` attribute tells HTMX which element should receive the `htmx-request` class. Without it, the class goes on the input itself (which isn't visible). With it, the class goes on the `.search-spinner` span, making it fade in during requests and fade out when the response arrives.

We use `opacity` instead of `display: none` so we can add a smooth CSS transition. But `opacity: 0` still takes up layout space, so we also set `position: absolute` on the spinner (with a `position: relative` parent) to pull it out of the normal flow. That way, there's no empty gap when the spinner isn't showing.

For a real application, you'd replace the "Searching…" text with a small animated spinner. You might also dim the existing results while new ones are loading (Exercise 6 covers this). The mechanism is always the same: `htmx-request` as a CSS hook, with `hx-indicator` controlling which element gets the class.

That's everything we need. Time to apply it to the profile site.

7.6 Hands-On: Adding Live Search to the Profile Site

Our profile site has profiles, a guestbook, and navigation between them. We're going to add a home page with a search bar that filters profiles as you type. This is the first feature that wouldn't be possible (or at least wouldn't be pleasant) with full page reloads.

Start from your Chapter 6 project:

```
cp -r chapter6 chapter7
cd chapter7
```

Your current folder structure:

```
chapter7/
├── .venv/
├── main.py
├── pyproject.toml
├── static/
│   └── style.css
└── templates/
    ├── base.html
    ├── guestbook.html
    ├── not_found.html
    └── profile.html
```

We need a few additions. A home page template, a partial template for search results, and some new profiles so there's enough data to make searching interesting:

```
chapter7/
├── .venv/
├── main.py
├── pyproject.toml
├── static/
│   └── style.css
└── templates/
    ├── base.html
    ├── guestbook.html
    ├── home.html
    ├── not_found.html
    ├── partials/
    │   └── profile_results.html
    └── profile.html
```

The `partials/` folder is a convention. Templates in here aren't full pages; they're HTML fragments meant to be swapped into an existing page by HTMX. There's nothing special about the folder name. It's just a way to keep your templates organized.

First, add the HTMX script to `base.html`. Put it in the `<head>` alongside your CSS link:

```html
<!-- templates/base.html -->
<!-- add inside <head>, after the CSS link -->
<script src="https://unpkg.com/htmx.org@2.0.4"></script>
```

Next, expand the profiles dictionary in `main.py`. Two profiles aren't enough for search to be useful. Add several more:

```python
# main.py
# replace the existing profiles dictionary

profiles = {
    "philipe": {
        "name": "Philipe Ackerman",
        "about": [
            "Hi! I'm a math professor, and a really good one at that.",
            "When I'm not teaching, you can find me cheering for my soccer team,
    Botafogo.",
        ],
        "skills": ["Math", "Having Long Hair", "Video Recording"],
    },
    "lucas": {
        "name": "Lucas",
        "about": [
            "Applied math student at UFRJ and backend developer.",
            "Part of the computing team at the Applied Computing League.",
        ],
        "skills": ["Python", "FastAPI", "Docker", "Linux"],
    },
    "zico": {
        "name": "Zico",
        "about": [
            "Applied math student at UFRJ who likes low-level programming.",
            "Part of the computing team at the Applied Computing League.",
        ],
        "skills": ["Rust", "C", "Python", "Git"],
    },
    "joao": {
        "name": "João",
        "about": [
            "Applied math student at UFRJ focused on data science.",
            "Part of the data science team at the Applied Computing League.",
        ],
        "skills": ["Python", "Pandas", "Machine Learning", "SQL"],
    },
    "eduardo": {
        "name": "Eduardo",
        "about": [
            "Applied math student at UFRJ who turns data into insights.",
            "Part of the data science team at the Applied Computing League.",
        ],
        "skills": ["Python", "Data Visualization", "R", "Statistics"],
    },
    "layza": {
        "name": "Layza",
        "about": [
            "Applied math student at UFRJ coordinating the league's projects.",
            "Part of the management team at the Applied Computing League.",
        ],
        "skills": ["Project Management", "Agile", "Data Analysis", "Leadership"],
    },
    "arthur": {
        "name": "Arthur",
        "about": [
            "Applied math student at UFRJ working on AI engineering.",
            "Part of the computing team at the Applied Computing League.",
        ],
        "skills": ["Python", "Machine Learning", "LLMs", "AI Engineering"],
    },
    "matheus": {
        "name": "Matheus",
        "about": [
            "Applied math student at UFRJ interested in product and strategy.",
            "Part of the management team at the Applied Computing League.",
        ],
        "skills": ["Product Management", "UX Research", "Strategy", "Analytics"],
    },
}
```

Now you need to build four things:

A GET endpoint at / that renders `home.html`. Pass the full profiles dictionary to the template so the page can display all profiles on first load.

A GET endpoint at `/search` that accepts an optional query parameter `q`. This endpoint filters profiles whose name contains the search term (case-insensitive) and returns the `partials/profile_results.html` template. When `q` is empty or missing, return all profiles. This endpoint returns a fragment, not a full page.

For the filtering logic, you'll want something like:

```
# inside the search endpoint
results = {
    username: profile
    for username, profile in profiles.items()
    if q.lower() in profile["name"].lower()
}
```

A home page template (`templates/home.html`) that extends `base.html`. It should contain a search input with HTMX attributes that sends requests to `/search` as the user types, a loading indicator, and a results `div` that initially shows all profiles. Set up the search input with `hx-get="/search"`, `hx-target="#results"`, `hx-trigger="keyup changed delay:300ms"`, and `hx-indicator`. The input's `name` attribute should be `q` to match the query parameter.

Here's a skeleton:

```
{% extends "base.html" %}

{% block title %}Home{% endblock %}

{% block content %}
<h1>Profiles</h1>

<div class="search-box">
    <!-- TODO: search input with HTMX attributes -->
    <!-- TODO: loading indicator -->
</div>

<div id="results">
    <!-- TODO: display all profiles initially -->
    <!-- use the same markup as the partial template -->
</div>
{% endblock %}
```

A partial template (`templates/partials/profile_results.html`) that renders a list of profile cards. This template does not extend `base.html` (it's a fragment, not a full page). Each card should show the profile's name, their first "about" line, their skills, and a link to `/profile/{username}`. This is the HTML that HTMX will swap into the `#results` div.

One important detail: the initial load of the home page and the HTMX search results should use the same markup for the profile cards. The cleanest way to do this is to use Jinja's `{% include %}` in `home.html` to embed the partial template on first load:

```
<div id="results">
    {% include "partials/profile_results.html" %}
</div>
```

This way, the initial page render and the HTMX updates produce identical HTML. The partial template just needs a `profiles` variable (a dictionary of username/profile pairs) in its context.

For the CSS, add styles to `static/style.css` for the search input (full width, decent padding), the profile cards (similar to the guestbook message cards), the loading indicator (hidden by default, visible during requests), and a "no results" state. You've styled cards and forms before. Apply what you know.

Test your work:

- Visit the home page. Do all six profiles appear?
- Type "lucas" in the search box. Do the results filter down to Lucas?
- Type "ar" in the search box. Does Arthur appear?
- Clear the search box. Do all profiles reappear?

- Open the Network tab. Do you see XHR requests going to `/search?q=...` as you type? Is the response an HTML fragment (no `<html>`, `<head>`, etc.)?
- Does the loading indicator appear briefly during search?
- Click a profile link from the search results. Does it navigate to the profile page?
- Navigate to the guestbook and back to home. Does everything still work?

When it's working, the experience should feel noticeably different from the full-page interactions in previous chapters. The profiles filter smoothly as you type, with no page flash and no loss of scroll position.

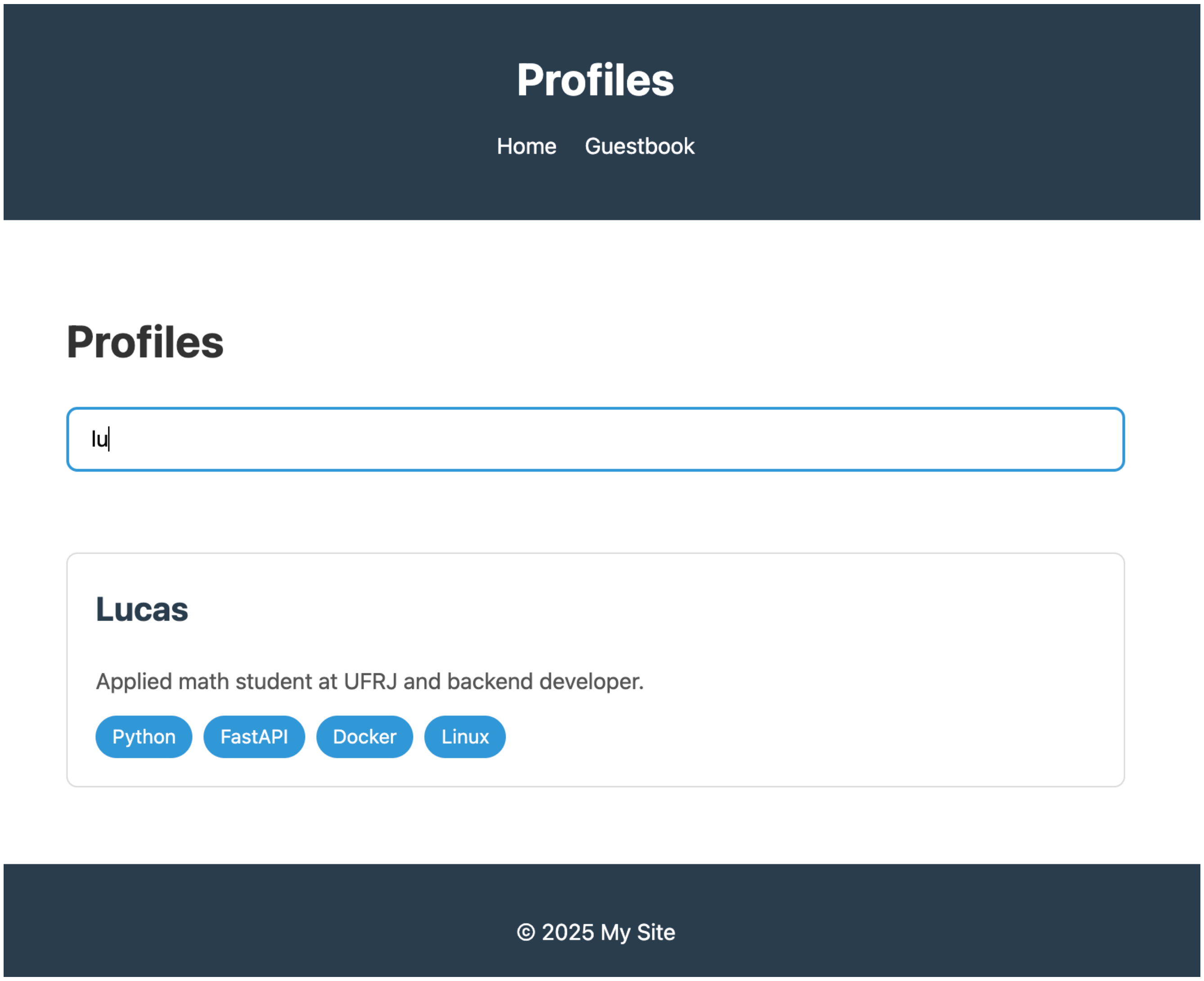


Figure 7.3: The profile site with live search

Stuck? Solution available

If you've given it an honest try and are truly stuck, a completed version of this project is available at github.com/Applied-Computing-League/practical-web-development/en/code/chapter7. Try to solve it yourself first. You learn more from the struggle than from reading the answer.

7.7 Chapter Summary

- Multi-page applications reload the entire page on every interaction. Single-page applications use JavaScript to update parts of the page. HTMX is a middle ground: server-rendered HTML with partial page updates.
- HTMX is controlled through HTML attributes, not JavaScript. Include it with a script tag and add attributes to your elements.
- `hx-get` and `hx-post` make HTTP requests when an element is interacted with. The server returns an HTML fragment, and HTMX swaps it into the page.
- `hx-target` controls where the response goes (a CSS selector). Without it, the response replaces the triggering element's content.
- `hx-swap` controls how the response is inserted: `innerHTML` (default, replace contents), `beforeend` (append), `outerHTML` (replace the element itself).
- `hx-trigger` controls what event fires the request. For live search: `keyup changed delay:300ms`.
- Partial templates (HTML fragments without `<html>`, `<head>`, etc.) are what the server returns for HTMX requests. Keep them in a `partials/` folder.
- Use `hx-indicator` and the `htmx-request` CSS class to show loading states during requests.

7.8 Exercises

1. The search currently filters by name only. Extend it to also match skills: if you type "python", it should return every profile that has "Python" in their skills list, even if "python" doesn't appear in their name. You'll need to update the filtering logic in the search endpoint. The template doesn't need to change.

2. Add an `hx-get` button to each guestbook message that loads an "edit form" in place. When clicked, the message text is replaced with a form (pre-filled with the current message). The form uses `hx-post` to save the edit and swaps the updated message back in. You'll need a GET endpoint that returns the edit form fragment and a POST endpoint that updates the message and returns the updated display fragment. This is the pattern for inline editing.

3. Add `hx-push-url="true"` to the search input. Type a search query and watch the browser's address bar. What happens? Now copy that URL and open it in a new tab. The search query should be in the URL, but does the page show filtered results? If not, update the home page endpoint to

accept an optional `q` parameter and pre-filter the profiles when the page first loads. This makes search results shareable.

4. The profile cards link to the profile page with a normal `<a>` tag, which triggers a full page load. Replace the link with an element that uses `hx-get="/profile/{username}"` and `hx-target="body"` to load the profile page without a full reload. What happens to the browser's back button? What happens to the URL? Read about `hx-push-url` in the HTMX docs and fix both issues. Think about whether this is actually better than a normal link.

5. Add a "load more" button to the home page. Show only the first three profiles initially, with a button at the bottom that fetches the next three via `hx-get`. The button should use `hx-swap="outerHTML"` so it replaces itself with the additional profiles (and a new "load more" button if there are still more). You'll need a server endpoint that accepts an `offset` parameter and returns the next batch of profile cards.

6. Right now, if the server is slow, the user sees "Searching..." but the old results stay visible. Add CSS that dims the results container while a search is in progress. You can do this with a parent wrapper that gets the `htmx-request` class via `hx-indicator`, and a CSS rule that reduces the opacity of the results div inside it. Test by adding `import time; time.sleep(1)` to the top of your search endpoint (remove it when you're done).

References

Gross, Carson. 2026. "Htmx Documentation." 2026. https://htmx.org.

Meta Open Source. 2026. "React Documentation." 2026. https://react.dev.

Mozilla Developer Network. 2026a. "CSS: Cascading Style Sheets." 2026. https://developer.mozilla.org/en-US/docs/Web/CSS.

———. 2026b. "HTML: HyperText Markup Language." 2026. https://developer.mozilla.org/en-US/docs/Web/HTML.

———. 2026c. "HTTP - HyperText Transfer Protocol." 2026. https://developer.mozilla.org/en-US/docs/Web/HTTP.

———. 2026d. "JavaScript Reference." 2026. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

Python Software Foundation. 2026. "Python Documentation." 2026. https://docs.python.org/3/.

Ramírez, Sebastián. 2026. "FastAPI Documentation." 2026. https://fastapi.tiangolo.com.

Ronacher, Armin. 2026. "Jinja2 Documentation." 2026. https://jinja.palletsprojects.com.