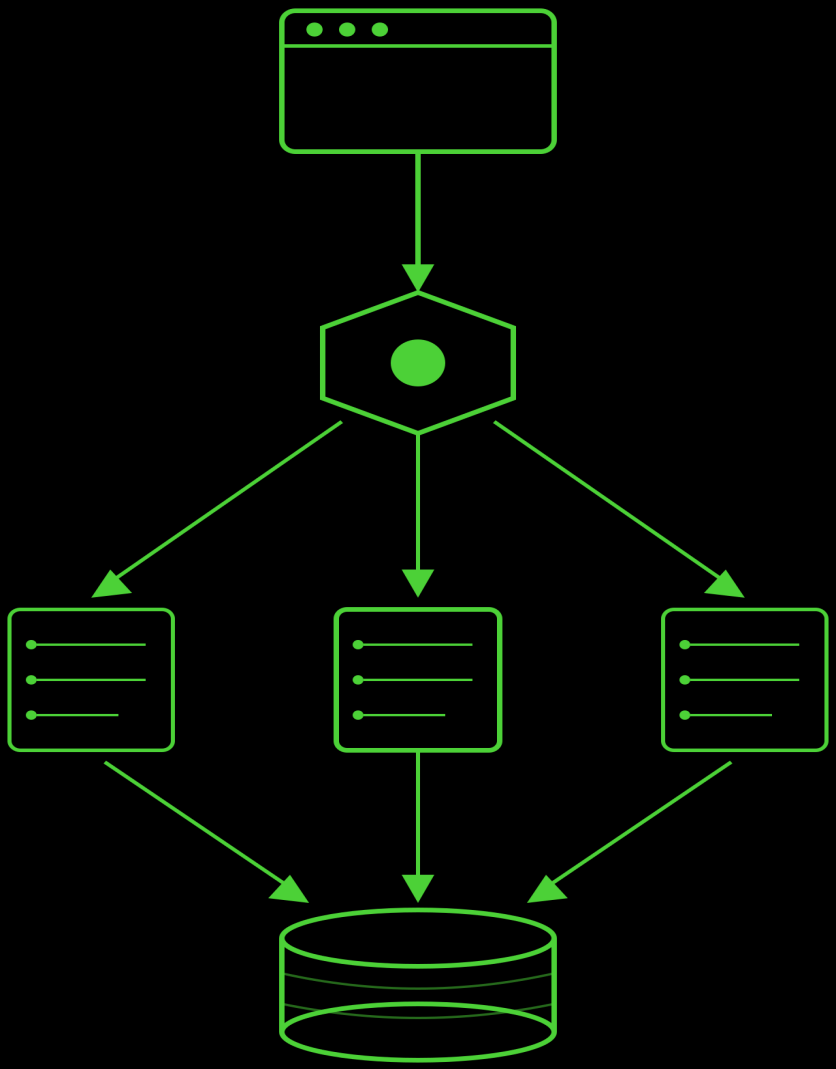
Practical Web Development

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). You'll understand both approaches and when each makes sense. We're not trying to cover every framework or teach you the "best" way. We're teaching you one coherent way that works, so you have solid ground to stand on when you explore other options later.

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

How to Use This Book

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

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

Each chapter follows a pattern:

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

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

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

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.

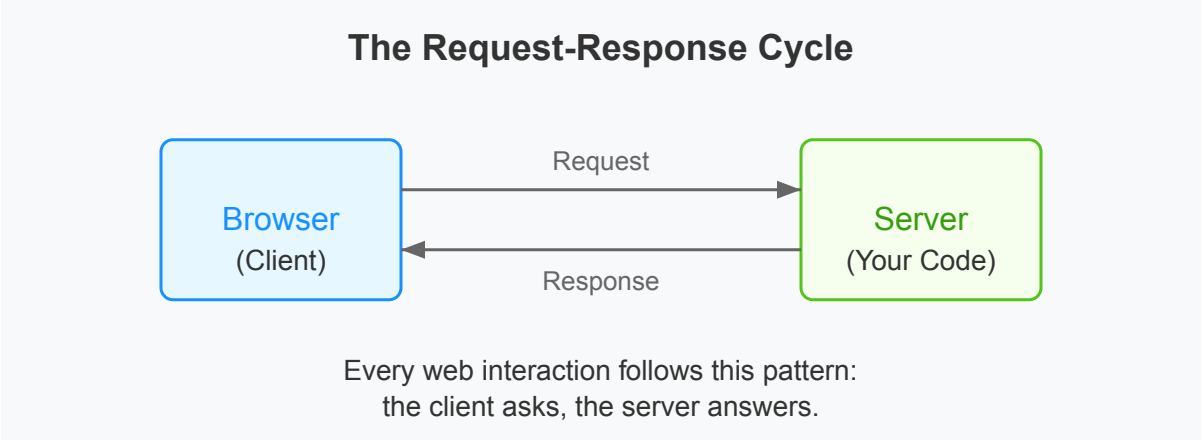


Figure 1.1: The Request-Response Cycle

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

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

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

1.2 HTTP: Requests and Responses

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

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

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

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

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

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

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

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

The server receives this request and sends back a response:

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

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

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

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

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

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

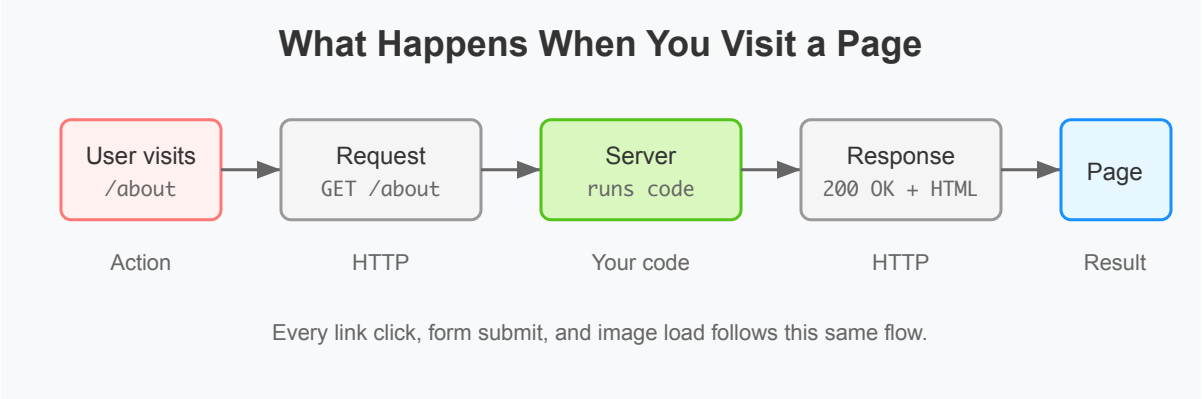


Figure 1.2: What Happens When You Visit a Page

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

1.3 URLs

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

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

Looking at it in detail:

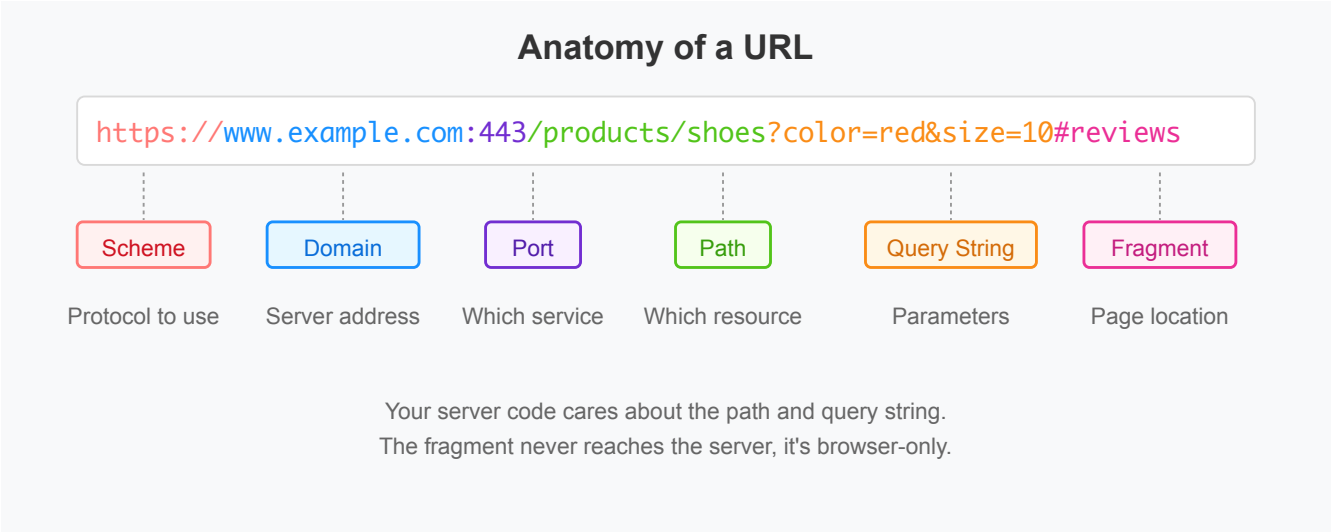


Figure 1.3: Anatomy of a URL

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

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

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

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

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

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

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

1.4 Static vs Dynamic Websites

Not all websites work the same way.

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

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

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

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

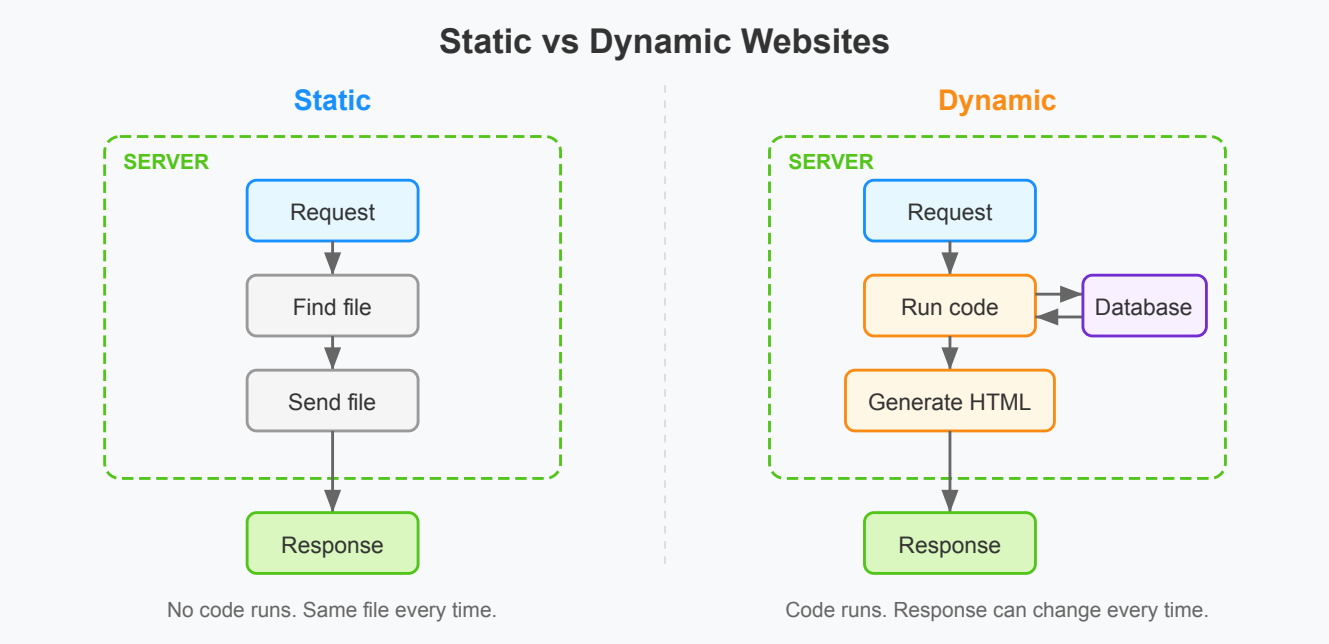


Figure 1.4: Static vs Dynamic Websites

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

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

1.5 Looking at Network Traffic

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

To open them:

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

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

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

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

1.6 What We're Building

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

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

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

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

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

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

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

1.7 Hands-On: Exploring HTTP with Browser Dev Tools

Let's put this into practice.

Exercise 1: Inspect a Page Load

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

Exercise 2: See a 404

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

Exercise 3: Query Parameters

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

Exercise 4: Watch Resources Load

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

1.8 Chapter Summary

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

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

1.9 Exercises

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

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"`.

Let's take a look at a complete HTML document and the relevant elements next.

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:

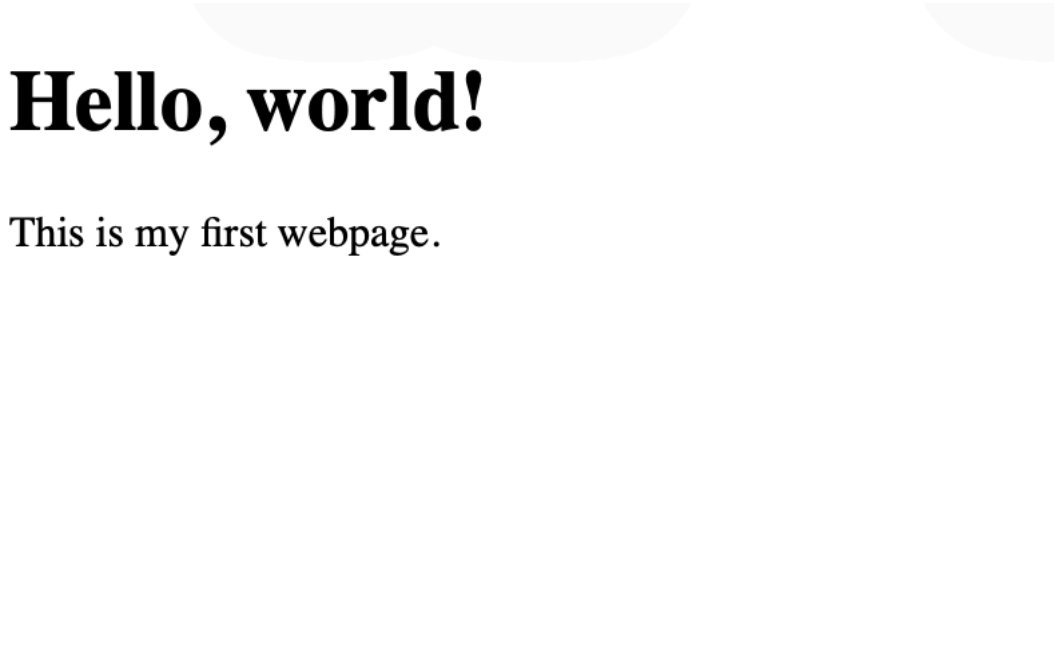


Figure 2.1: A basic HTML page rendered in the browser

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):

```
<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:

```
<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):

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

And ordered lists (numbered):

```
<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:

```
<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:

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


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

2.4 Semantic Markup

Early HTML was full of tags like <font>, , and <b>. These described how things should look, not what they meant. Modern HTML separates structure (HTML) from presentation (CSS).

Semantic tags describe the meaning of content:

```
<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).

You could build the same layout using only tags:

```
<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:

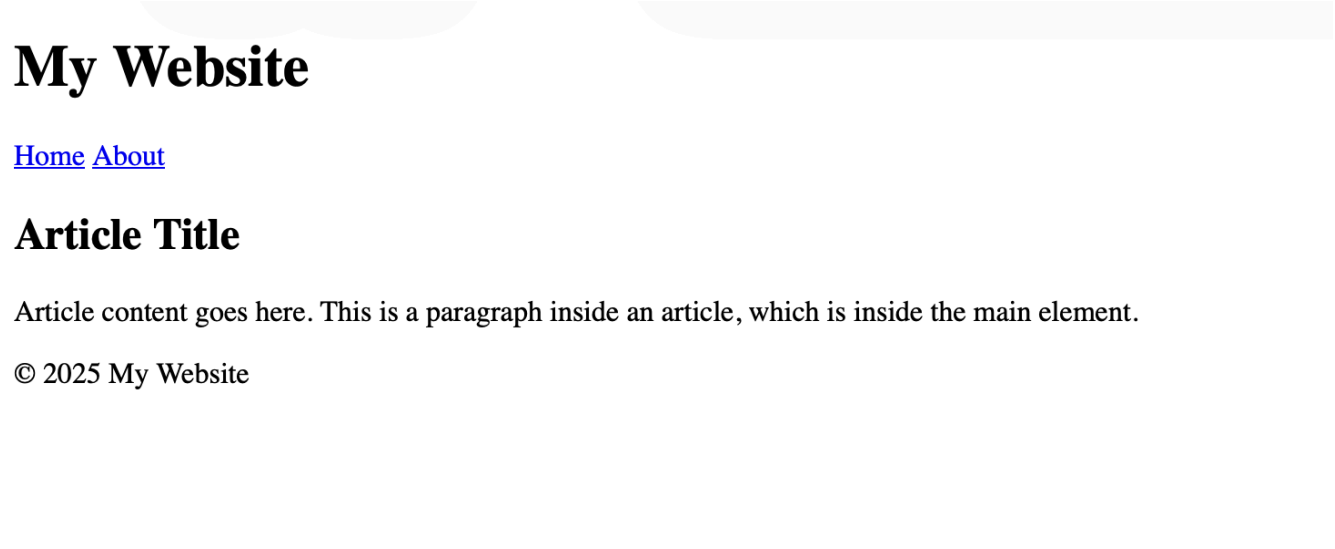


Figure 2.2: Semantic HTML elements rendered in the browser

Both approaches render the same way. But semantic tags help screen readers understand the page structure, help search engines identify important content (better for SEO), and make your code easier to read.

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. Screen readers and search engines care about this distinction.

2.5 Links and Navigation

Links are what make the web a web.

A basic link:

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

Linking to a page on the same site:

```
<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:

```
<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:

```
<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:

```
<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:

```
<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:

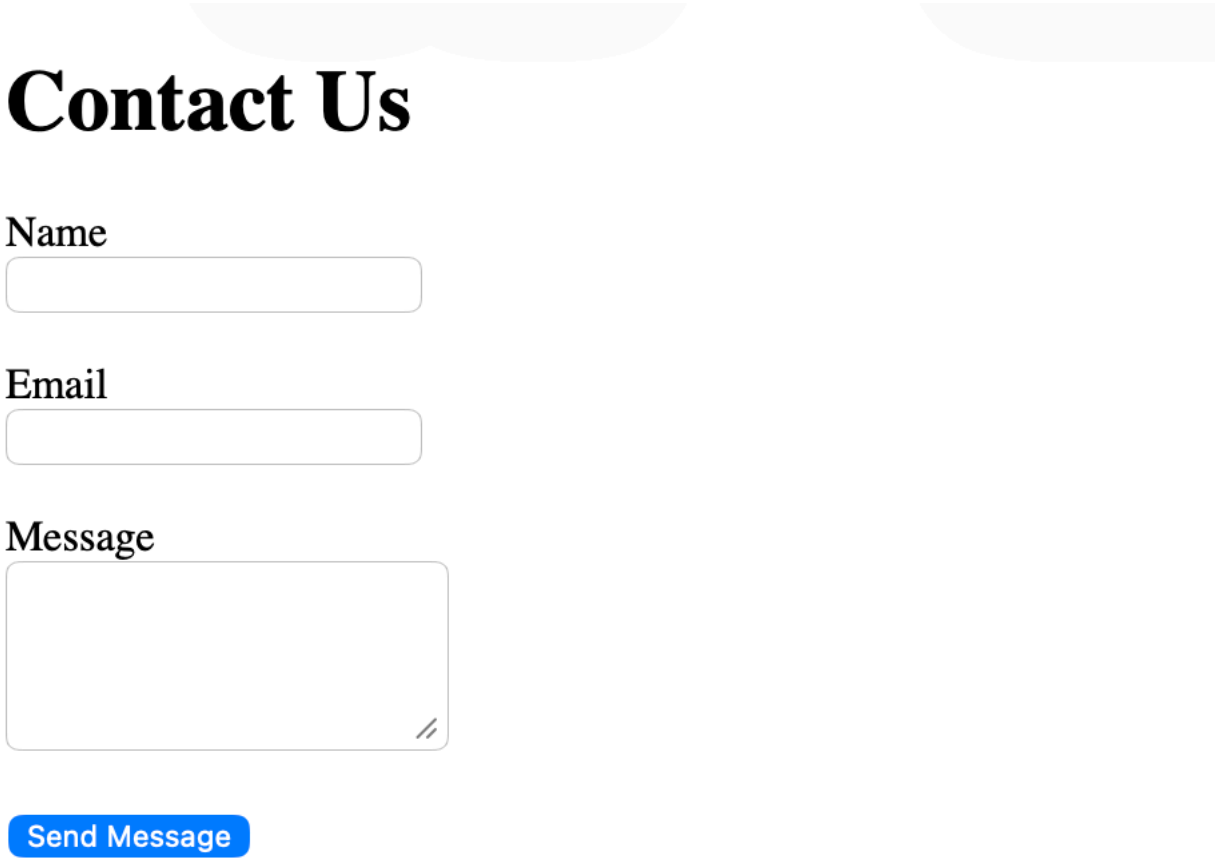


Figure 2.3: A contact form rendered in the browser

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:

```
<!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

Figure 2.4: The profile page template rendered in the browser

Ugly, I know. Bug, 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 to include a dropdown for "Reason for contact" with options like "General inquiry," "Job opportunity," and "Other."
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 three different websites. Look at how they structure their HTML. Can you find the `<header>`, `<main>`, and `<footer>` sections? How deeply nested are their 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. Now let's see how to actually connect the two and apply CSS styles to our HTML structured text.

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, but we'll focus on the ones you'll see constantly.

Colors:

```
h1 {
    color: navy;                /* text color */
    background-color: #f0f0f0;  /* background */
}
```

Colors can be names (`red`, `navy`, `white`), hex codes (`#ff0000`, `#f0f0f0`), or RGB values (`rgb(255, 0, 0)`). Hex codes are most common.

Typography:

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

h1 {
    font-size: 32px;
    font-weight: bold;
}
```

```
.subtle {
    font-style: italic;
    color: #666;
}
```

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

`line-height` controls spacing between lines. A unitless number like `1.5` means 1.5 times the font size.

Spacing:

```
p {
    margin: 20px;     /* space outside the element */
    padding: 10px;    /* space inside the element */
}
```

The difference between margin and padding matters once you understand the box model (next section).

You can set each side separately:

```
p {
    margin-top: 20px;
    margin-bottom: 20px;
    margin-left: 0;
    margin-right: 0;
}

/* shorthand: top right bottom left */
p {
    margin: 20px 0 20px 0;
}

/* shorthand: vertical horizontal */
p {
    margin: 20px 0;
}
```

Borders:

```
.card {
    border: 1px solid #ccc;
    border-radius: 8px;          /* rounded corners */
}
```

Size:

```
img {
    width: 200px;
    height: auto;                /* maintain aspect ratio */
}
```

```css
.container {
    max-width: 800px;           /* won't grow larger than this */
}
```

Text:

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

a {
    text-decoration: none;      /* removes underline */
}

.uppercase {
    text-transform: uppercase;
}
```

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 others

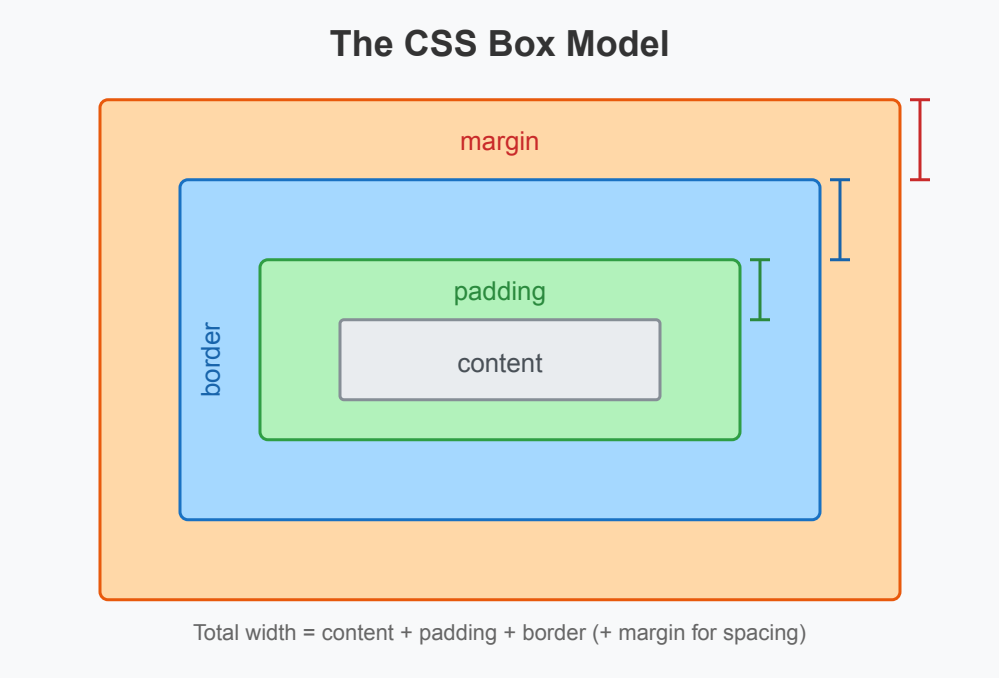


Figure 3.1: The CSS Box Model

Here's an example:

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

How wide is this element on the page? It's not 300px. By default, `width` sets only the content width. The total width is composed of these:

- Content: 300px
- Padding: 20px left + 20px right = 40px
- Border: 2px left + 2px right = 4px
- Total: 344px (margin doesn't count toward the element's size, but does affect spacing)

This math is annoying, but there's a fix:

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

With `box-sizing: border-box`, `width` includes padding and border. Now if you set `width: 300px`, the element is actually 300px wide, and the browser subtracts space for padding and border

from the content area.

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

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, but it can be confusing (CSS is often a pain, get used to it).

3.5 Layout with Flexbox

Getting elements to sit next to each other used to be more painful. CSS had `float`, which was designed for wrapping text around images and was hacked to do layout. It was confusing and error-prone.

Flexbox (Flexible Box Layout) was designed specifically for layout. It makes things like centering, spacing, and alignment easy.

To use flexbox, set `display: flex` on a container:

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

The css:

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

The links now sit side by side instead of stacking vertically. The container is a "flex container" and its children are "flex items."

justify-content controls horizontal alignment:

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

You have some options for it:

- `flex-start`: items at the start (default)
- `flex-end`: items at the end
- `center`: items in the center
- `space-between`: items spread out, first at start, last at end
- `space-around`: items spread out with space around each
- `space-evenly`: items spread out with equal space between

align-items controls vertical alignment:

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

There are also a few options:

- `stretch`: items stretch to fill container height (default)
- `flex-start`: items at the top
- `flex-end`: items at the bottom
- `center`: items vertically centered

gap adds space between items:

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

This is cleaner than adding margin to each item.

flex-direction controls which way items flow:

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

And the options:

- `row`: left to right (default)
- `column`: top to bottom
- `row-reverse`: right to left
- `column-reverse`: bottom to top

When you change direction, `justify-content` and `align-items` swap roles. In a column layout, `justify-content` controls vertical spacing and `align-items` controls horizontal alignment.

Centering with flexbox is finally easy:

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

This centers the content both horizontally and vertically. Before flexbox, this was a lot harder to do.

Here's a common page layout:

```
.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 `flex: 1` on `main` tells it to grow and fill whatever space is left after header and footer. This keeps the footer at the bottom even when there's not much content.

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:

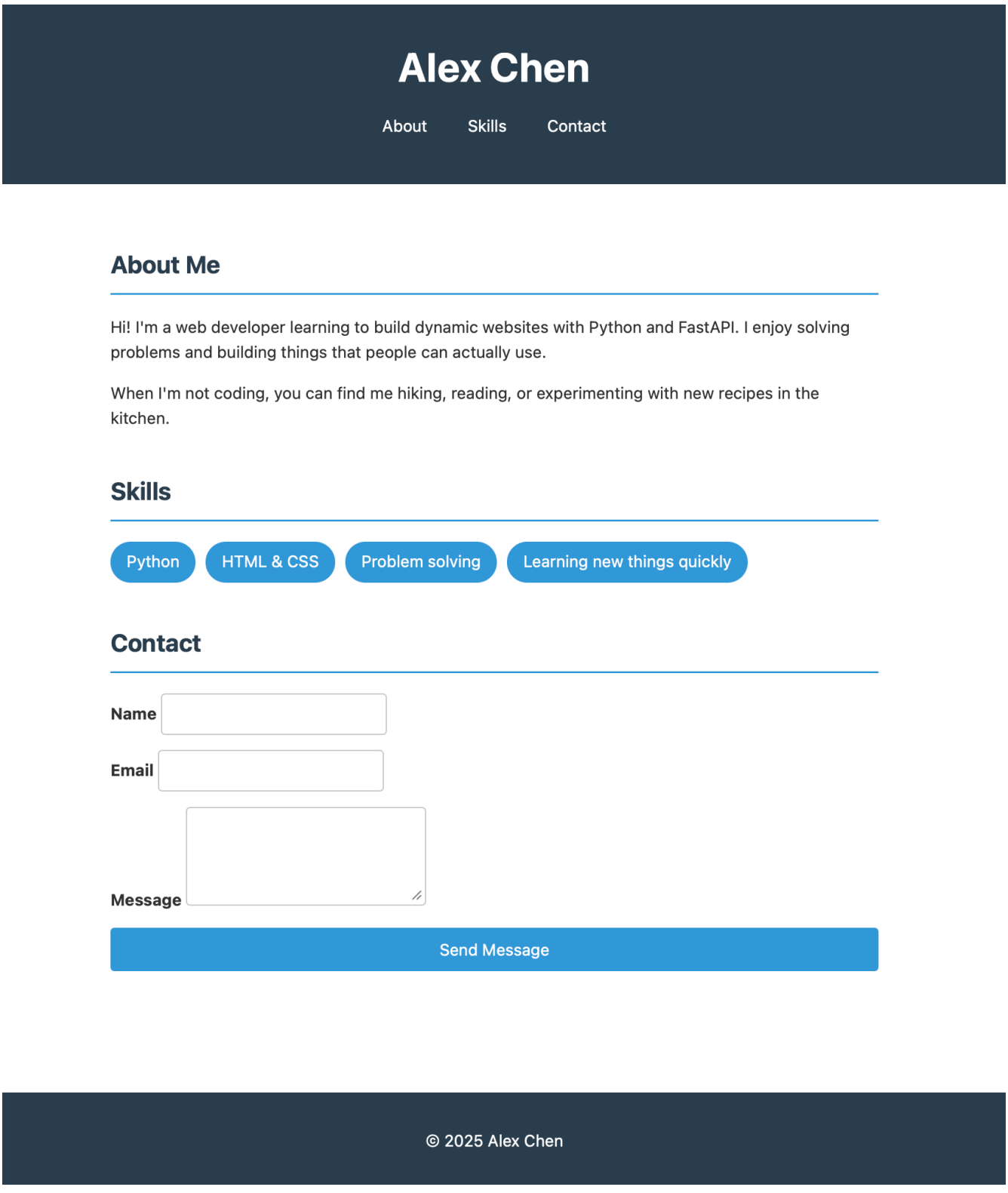


Figure 3.2: The styled profile page

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. Change the color scheme of your profile page. Pick a different primary color and update the header, footer, links, and buttons to match.
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 instead of plain text links. Add background colors, padding, and rounded corners.
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. Each card should have a title, description, and a "View Project" link. Use flexbox to arrange them in a row, and make them wrap 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.

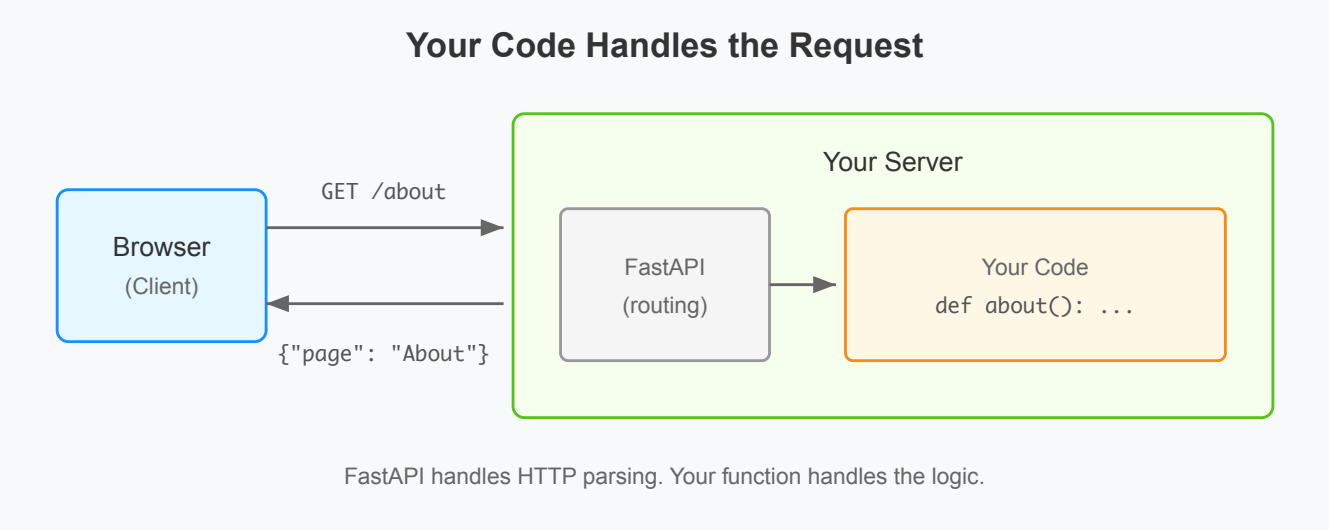


Figure 4.1: Your code handles the request

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. But before we can write any code, we need to set up our project.

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:

```
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:

```
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:

```
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:

```
# 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:

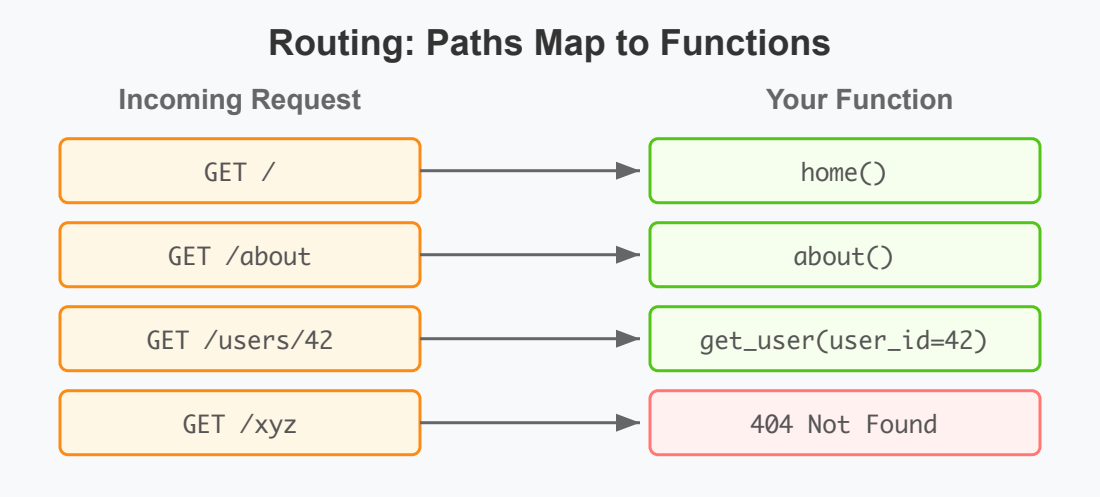


Figure 4.2: Routing maps paths to functions

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.

```python
# 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:

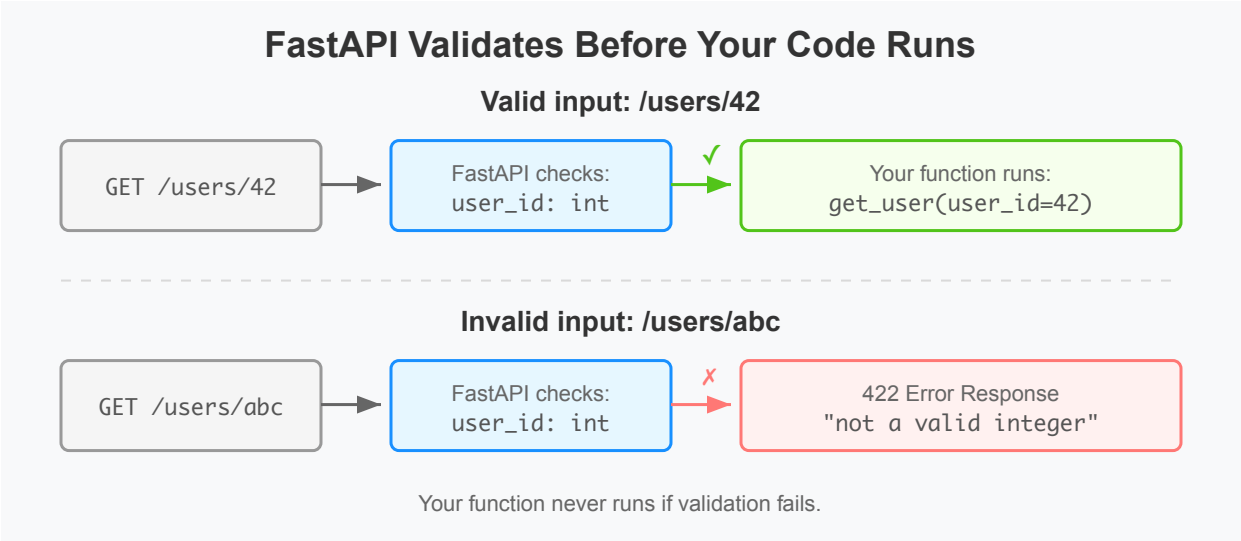


Figure 4.3: FastAPI validates before your code runs

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:

```python
@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. But path parameters aren't the only way to pass data to your server. Sometimes you want optional filters or settings that don't belong in the URL path itself.

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.

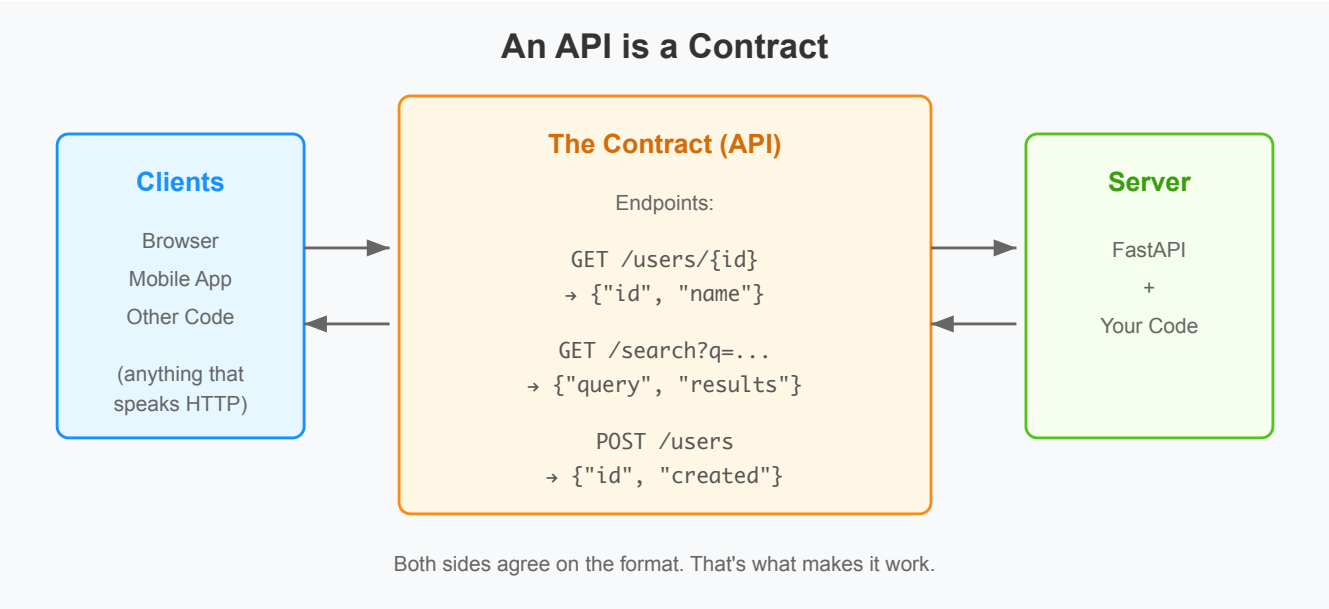


Figure 4.4: An API is a contract

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.

So far, all our endpoints return dictionaries, which FastAPI converts to JSON:

```
{"user_id": 1, "name": "User 1"}
```

The `Content-Type` header is set to `application/json`. That's an API response. But for a website that humans visit directly, we want to return HTML instead.

4.9 Returning HTML

For a website that humans visit (not just an API for code to consume), we want to return HTML that browsers can render. First, let's delete everything we had in our `main.py`, then we need to import `HTMLResponse`:

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

Then we can return HTML strings from our endpoints:

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

@app.get("/", response_class=HTMLResponse)
def home():
    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`. Now the browser renders the HTML instead of showing raw text.

Writing HTML inside Python strings gets messy fast. In the next chapter, we'll use templates to keep HTML in separate files. But this shows the basic idea: your function returns something, FastAPI sends it to the browser with the right headers.

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.

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` step by step. 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. Right now it's hardcoded, but it could come from a database, a file, or user input.

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:

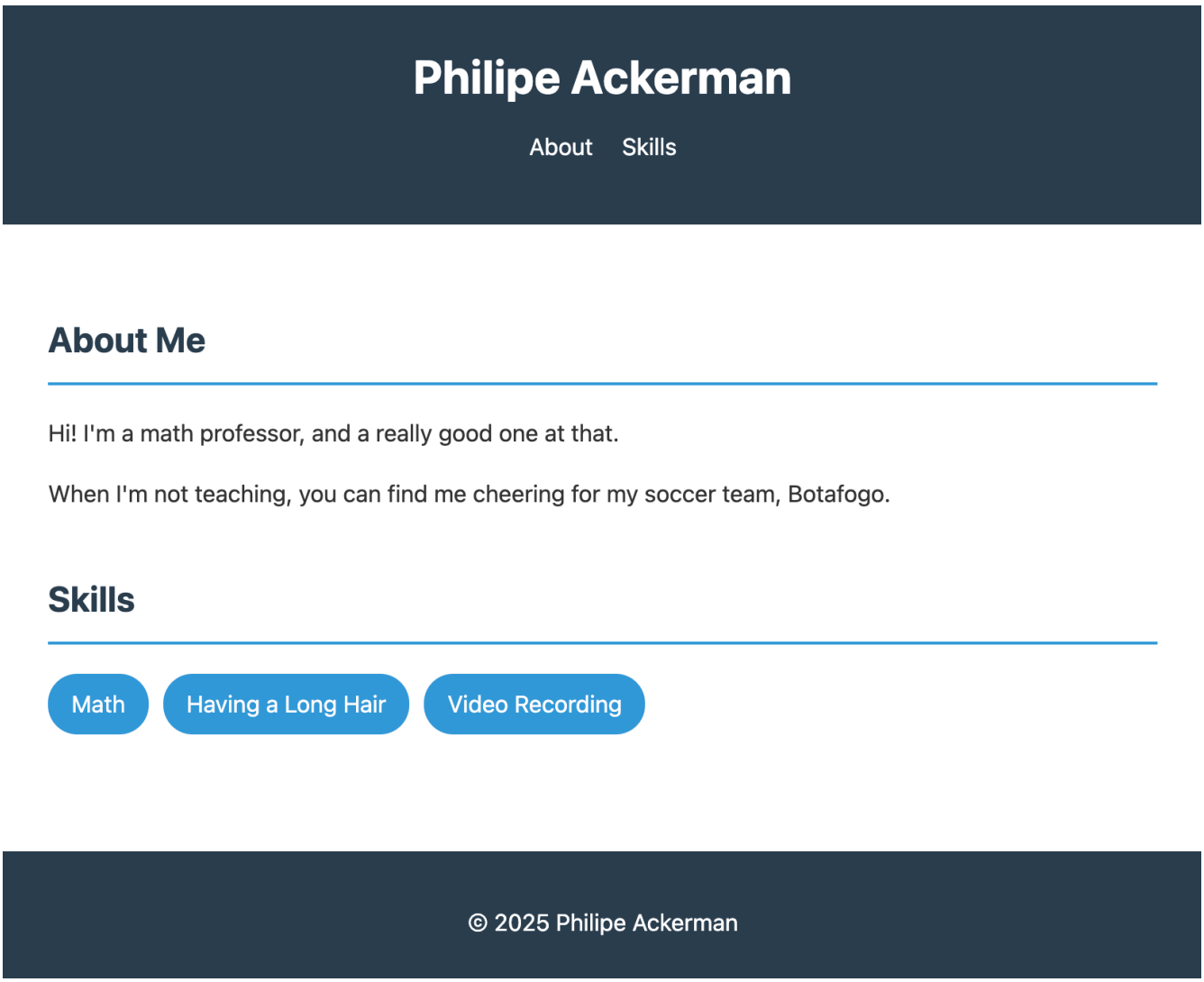


Figure 4.5: The profile page served by FastAPI

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:

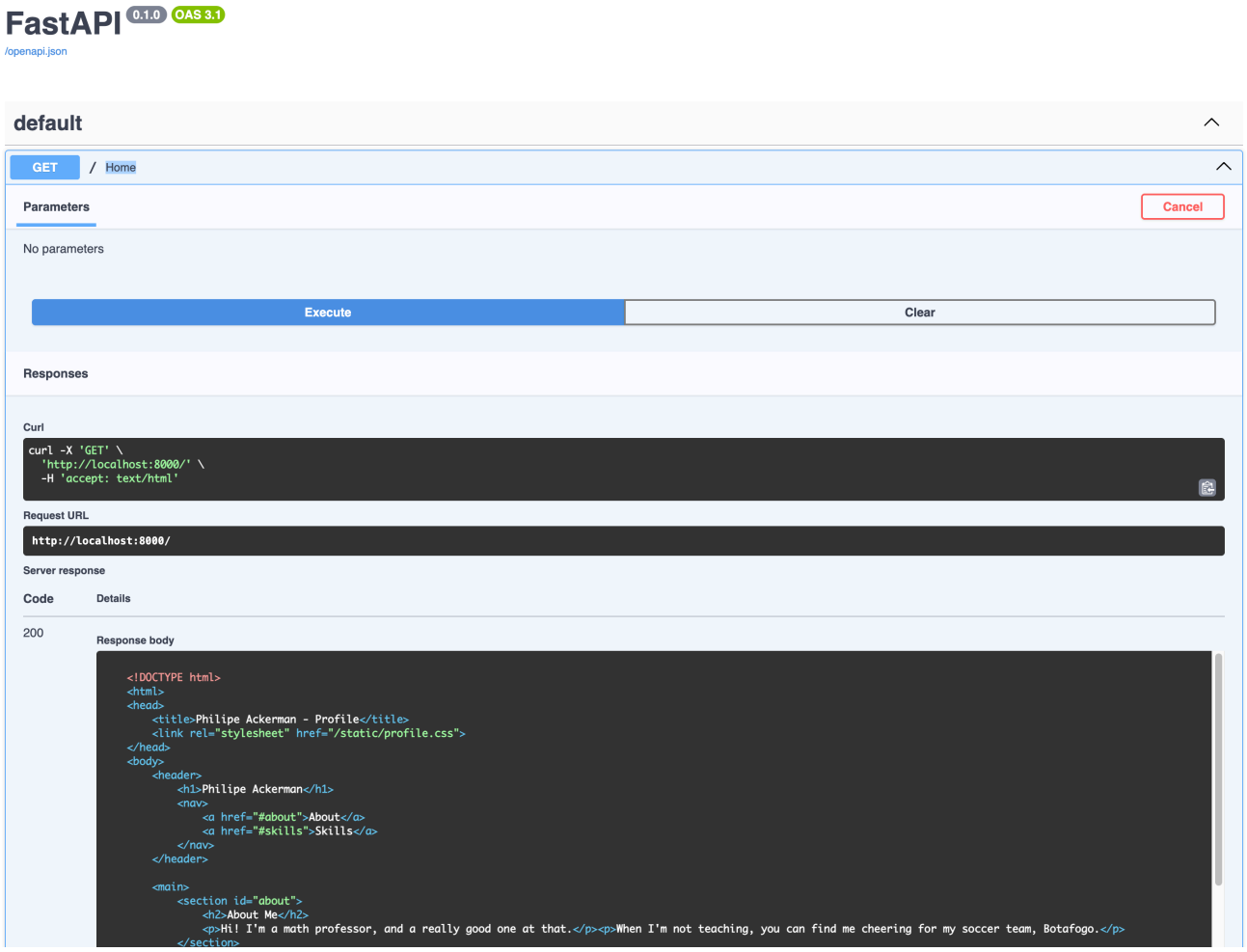


Figure 4.6: FastAPI generates interactive documentation

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:

```
@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:

```
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:

```
<!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`:

```
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:

```
# 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:

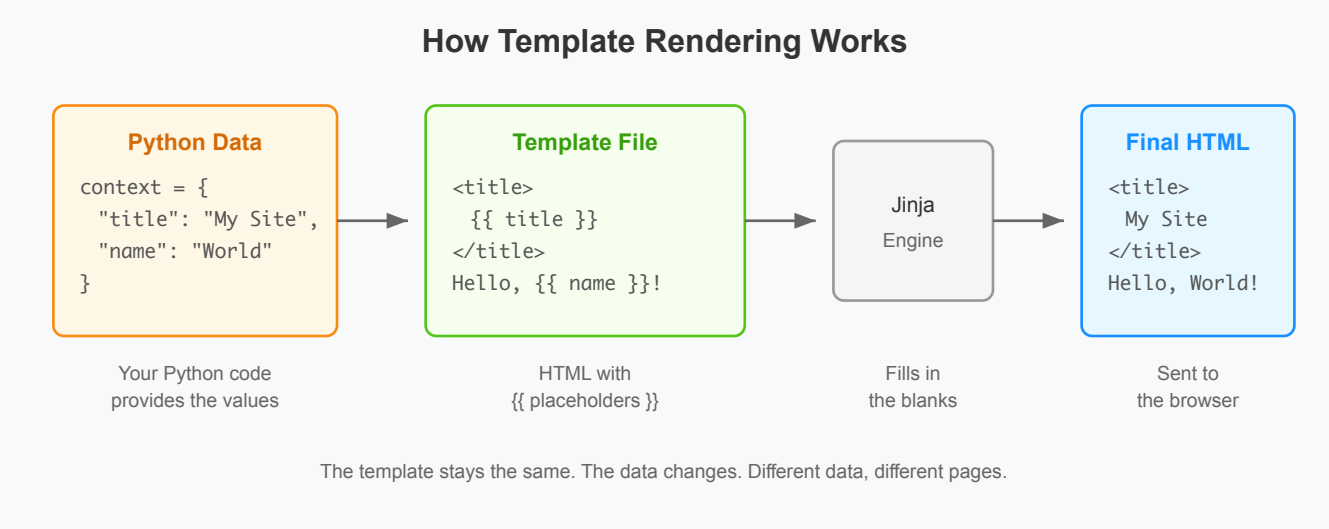


Figure 5.1: How template rendering works

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:

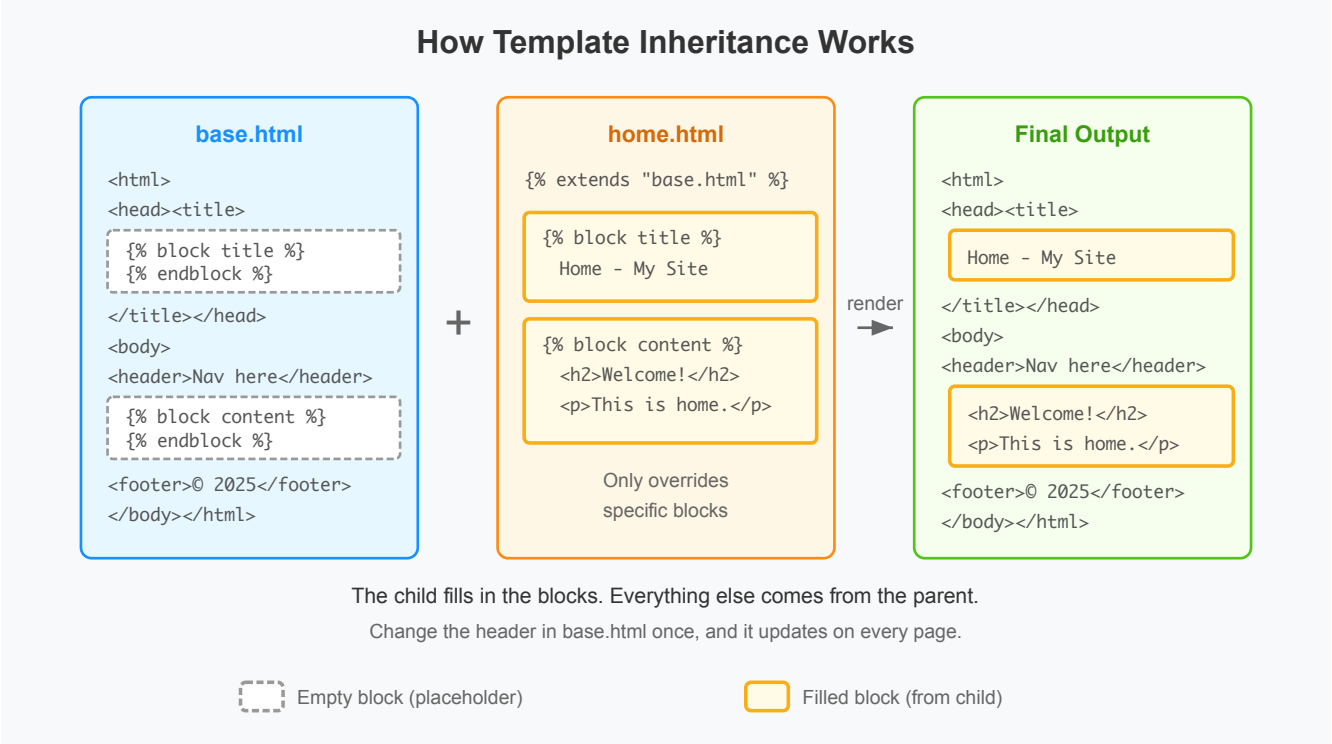


Figure 5.2: How template inheritance works

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:

```
<!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:

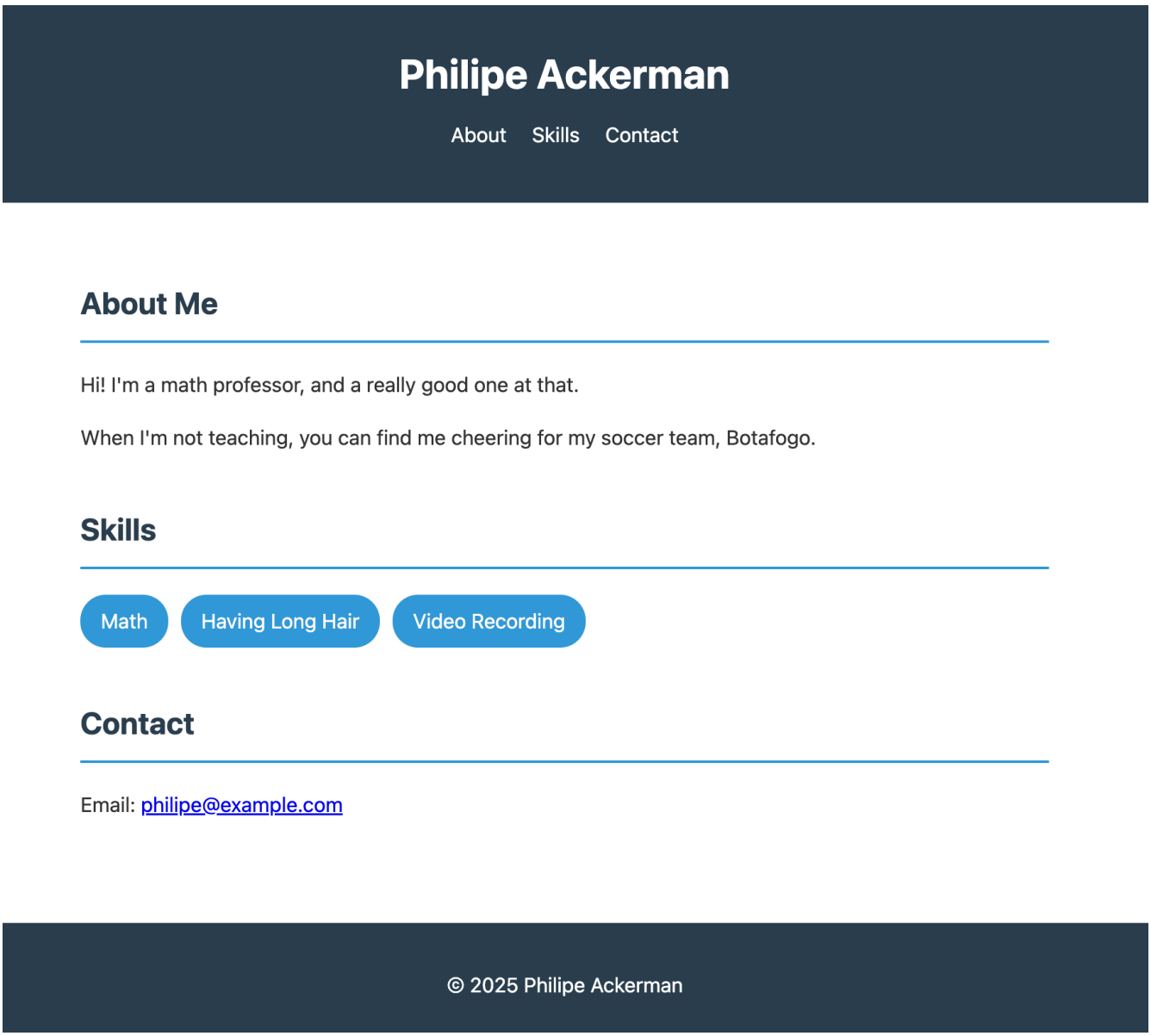


Figure 5.3: The profile page using templates

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",
    },
    "maria": {
        "name": "Maria Santos",
        "about": [
            "Software developer by day, guitarist by night.",
            "I believe code should be as elegant as music.",
        ],
        "skills": ["Python", "FastAPI", "Guitar", "Coffee Brewing"],
        "email": "maria@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:

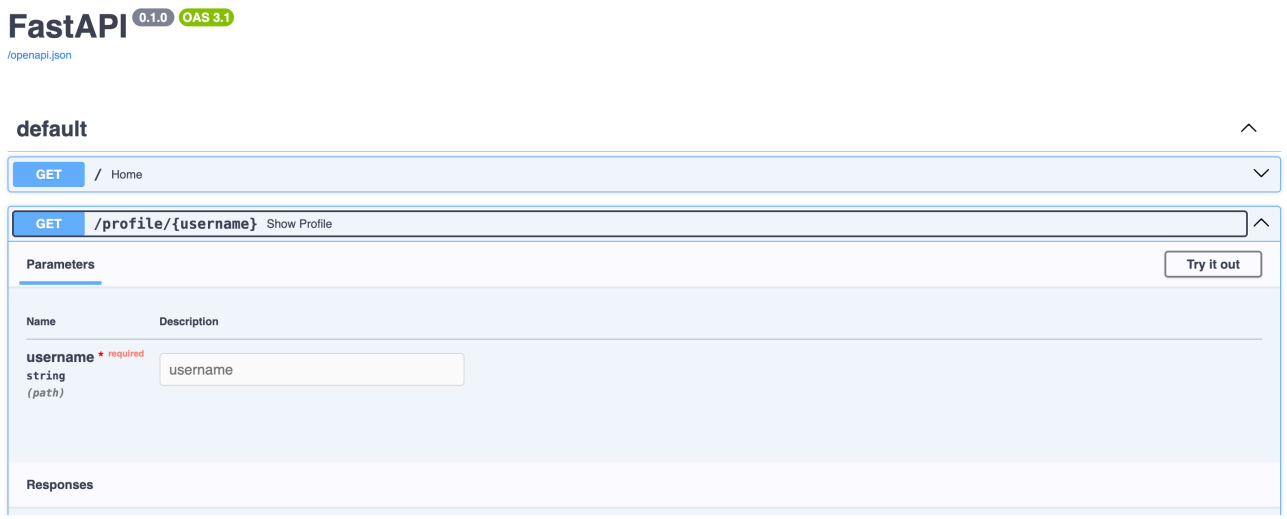


Figure 5.4: The endpoint expanded with input field ready

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

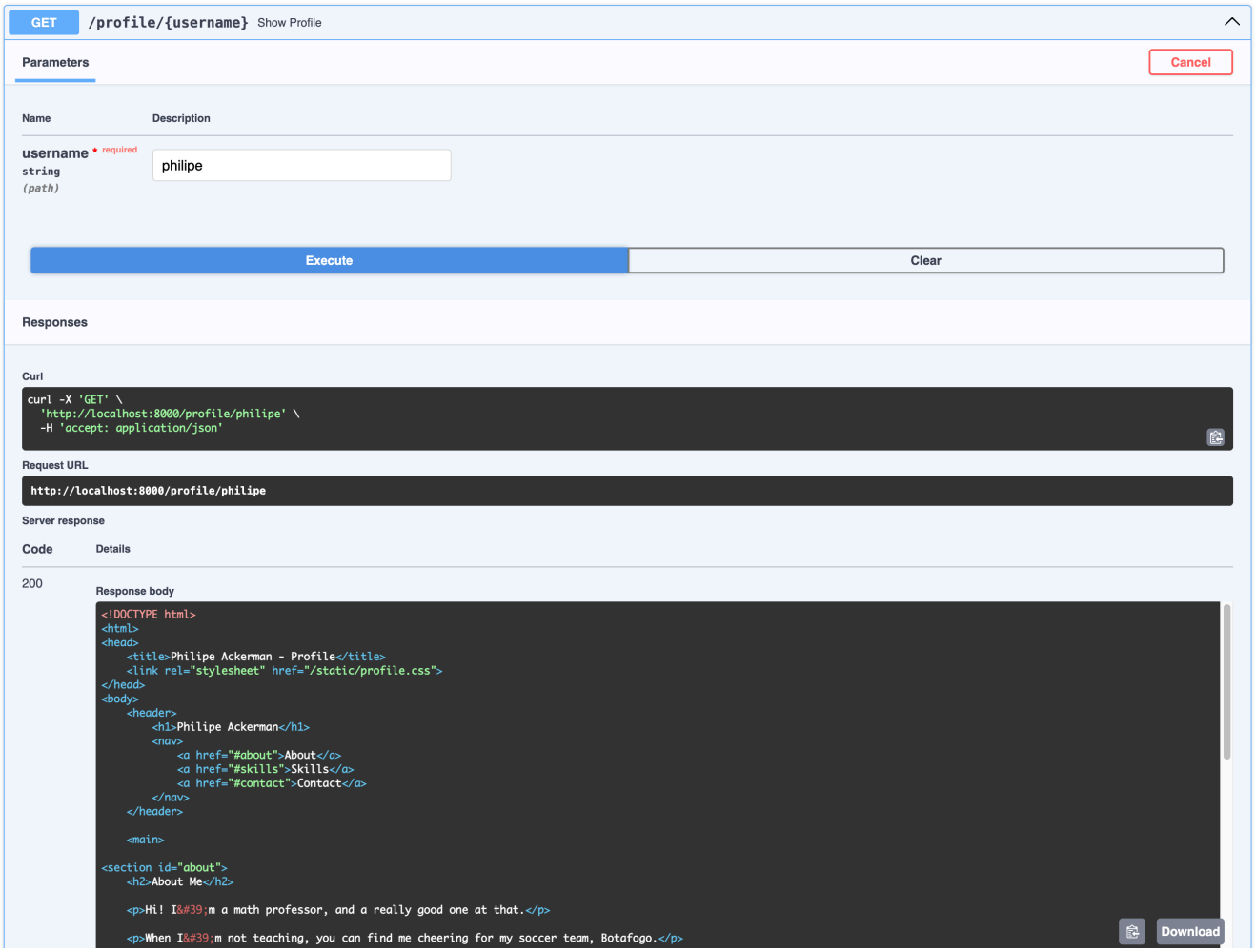


Figure 5.5: The response after executing the request

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
@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.

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.