Practical Web Development

From Server-Rendered Pages to React with Python and FastAPI

Igor Benav

Contents

Preface

To be written.

About This Book

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

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

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

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

How to Use This Book

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

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

Each chapter follows a pattern:

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

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

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

Prerequisites

You'll need:

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

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

1 Chapter 1: How the Web Works

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

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

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

1.1 Clients and Servers

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

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

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

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

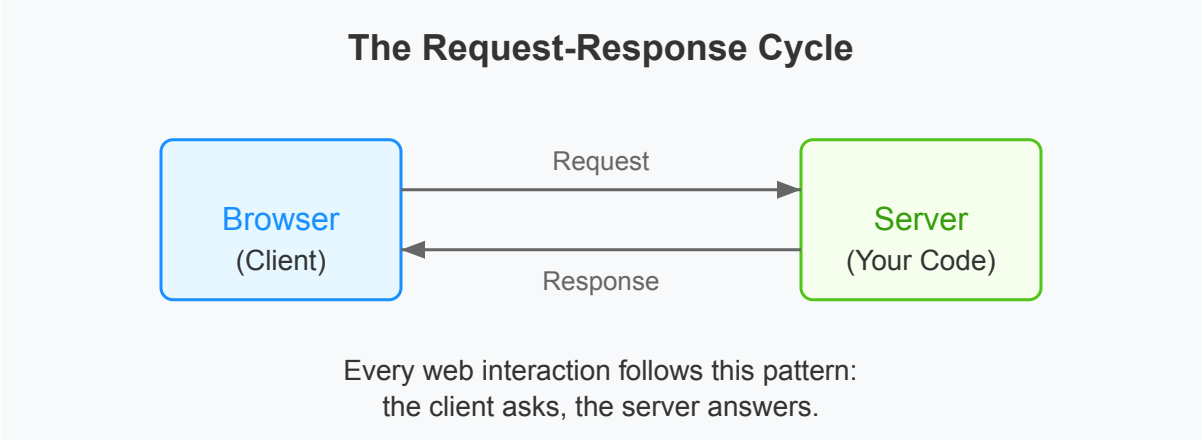

Figure 1.1: Figure 1.1: The Request-Response Cycle

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.

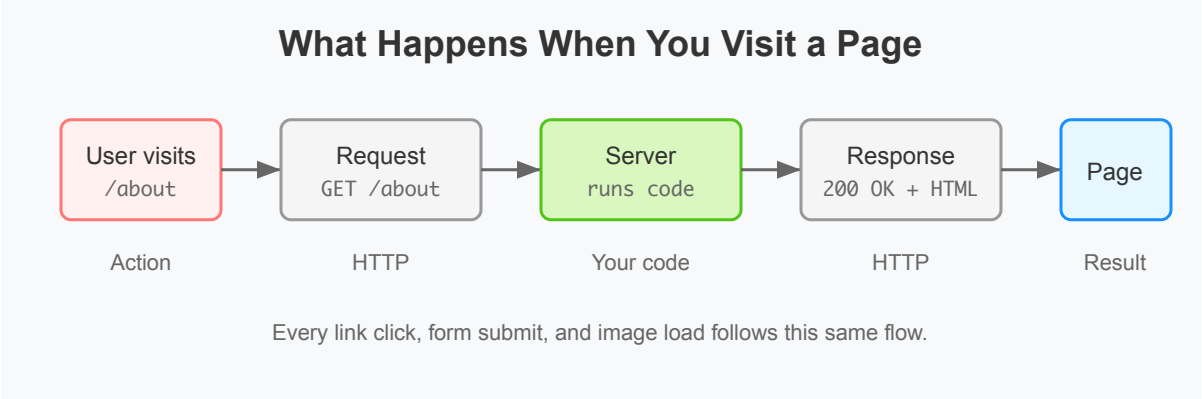

Figure 1.2: Figure 1.2: What Happens When You Visit a Page

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:

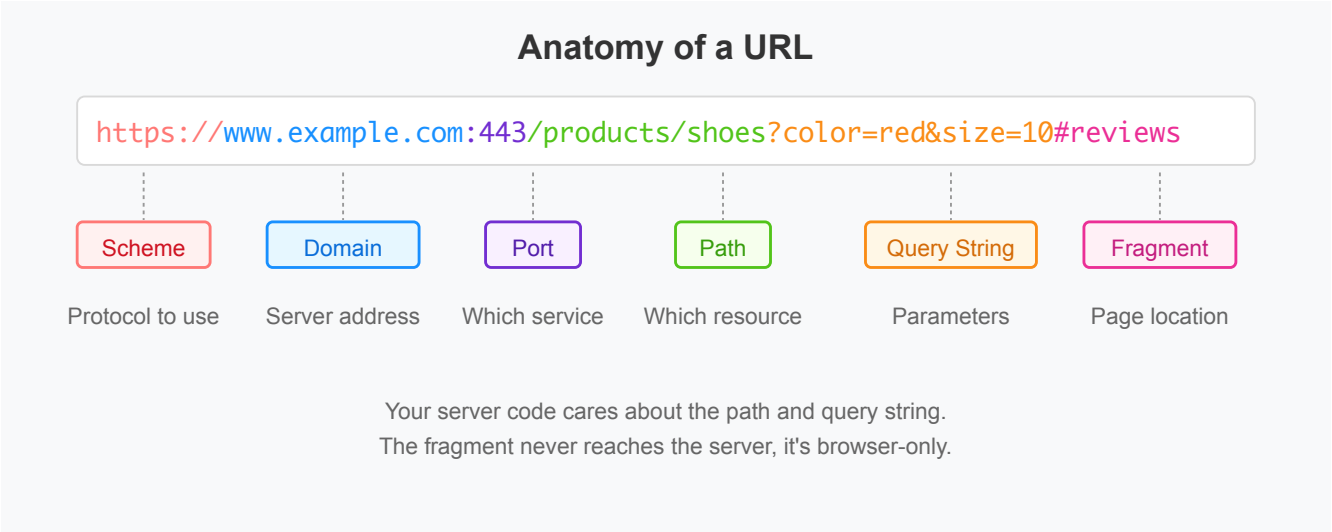

Figure 1.3: Figure 1.3: Anatomy of a URL

`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.

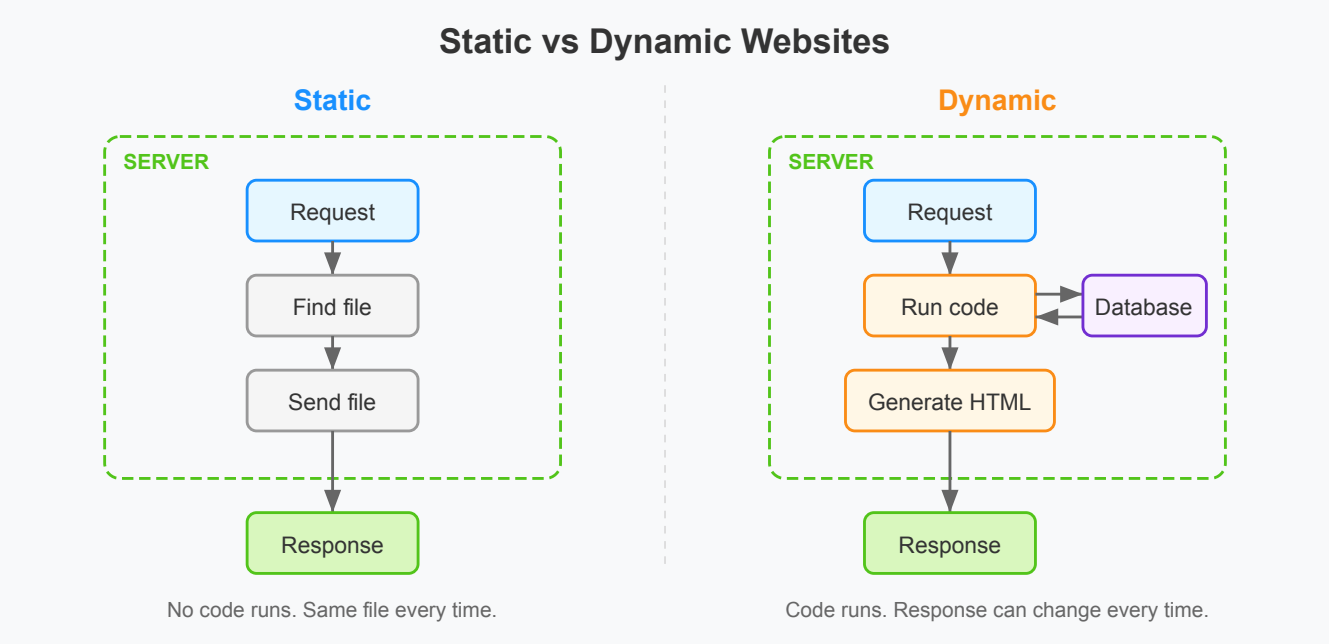

Figure 1.4: Figure 1.4: Static vs Dynamic Websites

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:

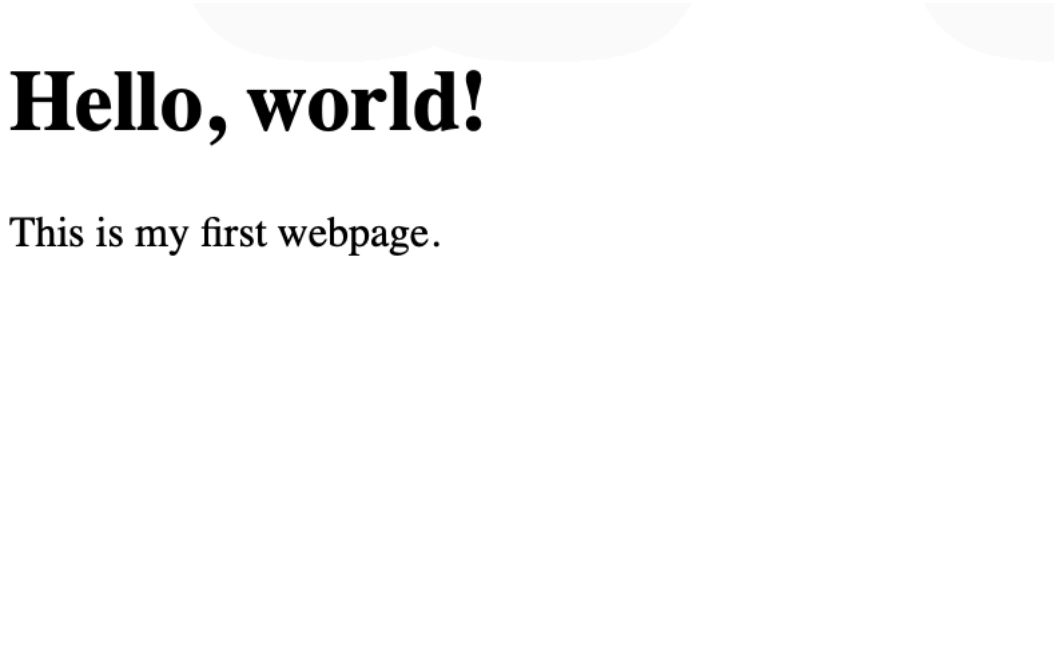

Figure 2.1: Figure 2.1: A basic HTML page rendered in the browser

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:

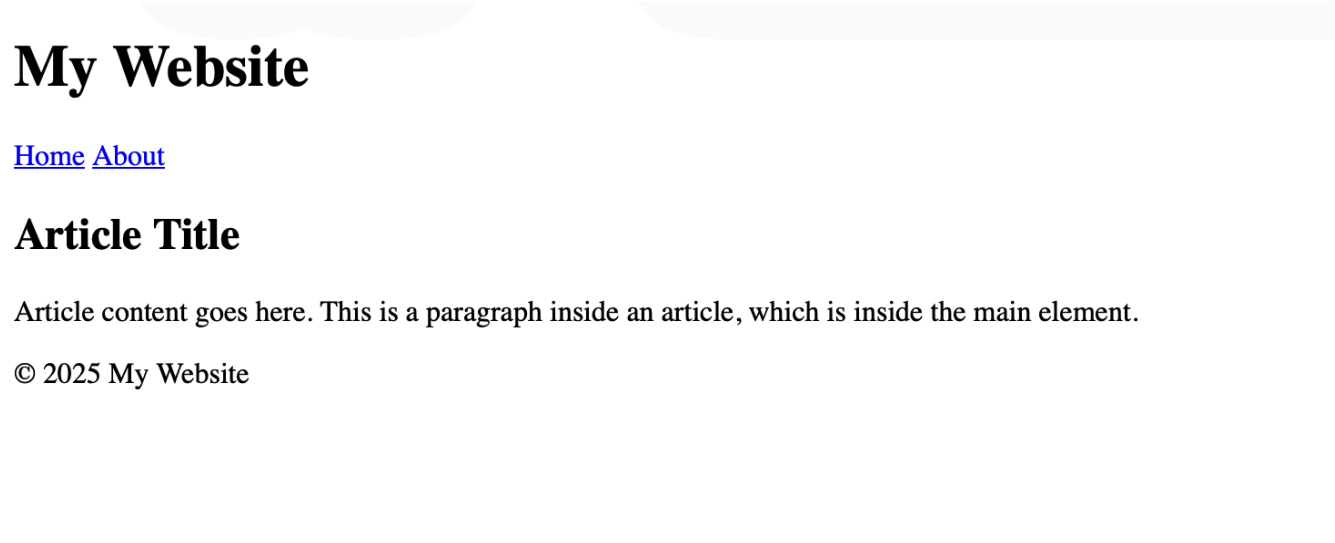

Figure 2.2: Figure 2.2: Semantic HTML elements rendered in the browser

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:

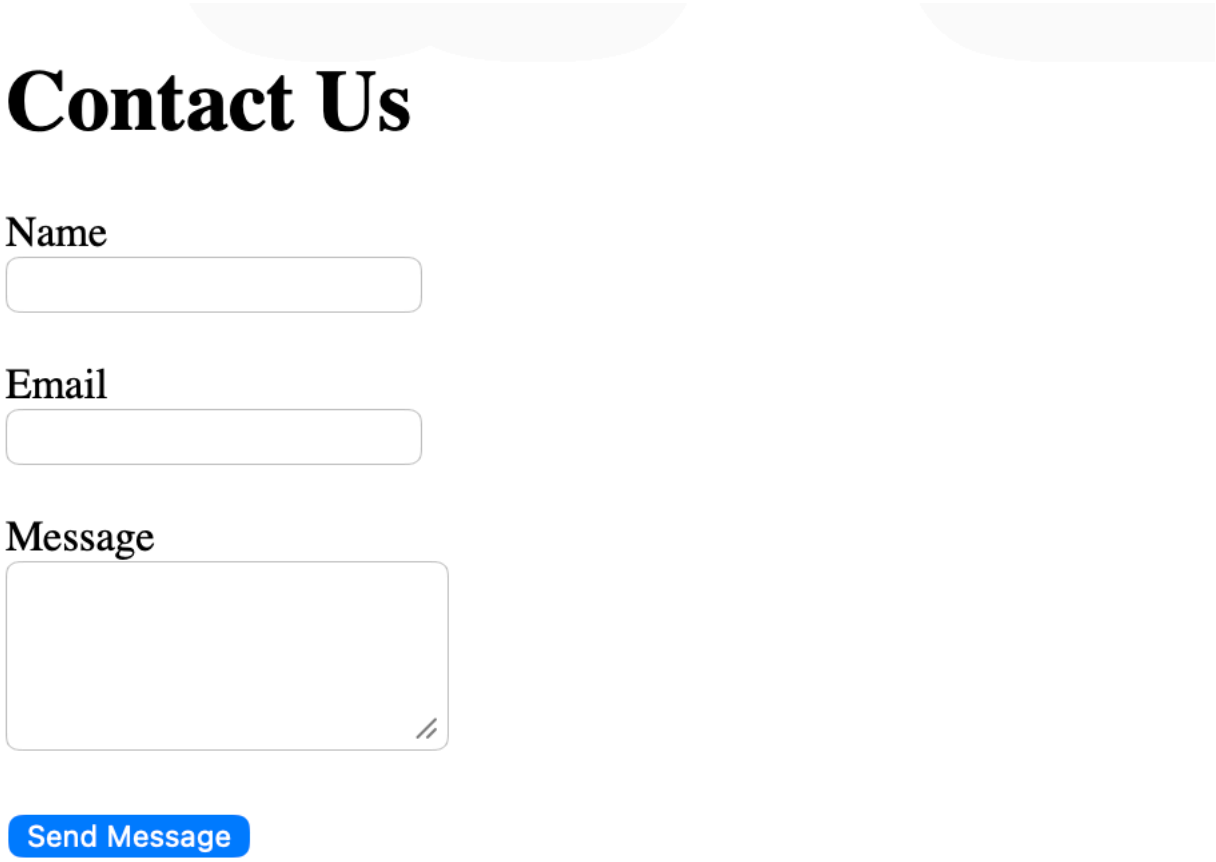

Figure 2.3: Figure 2.3: A contact form rendered in the browser

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.

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: 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.)

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.