

# Web Fundamentals & HTML Essentials

## TABLE OF CONTENTS

1. Introduction
2. Learning Objectives
3. Web Fundamentals
  - 3.1 Client-Server Model
  - 3.2 HTTP Basics
  - 3.3 HTTP Headers
  - 3.4 Cookies and Sessions
  - 3.5 Caching on the Web
  - 3.6 Cross-Origin Resource Sharing (CORS)
  - 3.7 Lab: Inspecting HTTP with Command-Line
4. HTML Essentials
  - 4.1 HTML Syntax and Structure
  - 4.2 Semantic HTML Elements
  - 4.3 HTML Forms
  - 4.4 Accessibility Basics
  - 4.5 Lab: Building a Semantic Homepage
  - 4.6 Intermediate Exercises
5. What We Have Learned
6. Brief Recap

## INTRODUCTION

The World Wide Web has become an integral part of modern life, and understanding how it works is essential for anyone entering web development. This seminar document provides a comprehensive overview of web fundamentals and HTML essentials, laying the groundwork for building and understanding websites. We will explore how web browsers and servers communicate, the role of the HTTP protocol, and the foundational structure of web pages using HTML.

In this unified segment, you will first learn about *Web Fundamentals* – the underlying mechanics that enable web interactions (such as client-server architecture and HTTP). We will then delve into *HTML Essentials*, covering how to structure content on the web using HyperText Markup Language. Throughout the document, you will find examples, exercises, and lab activities to practice each concept. By mastering these fundamentals, you'll be prepared to progress to more advanced topics in web development.

## LEARNING OBJECTIVES

After completing this seminar, you should be able to:

- **Explain** the client-server model of the web and how browsers (clients) and web servers interact.
- **Describe** the purpose of the HTTP protocol, including requests, responses, status codes, and headers.
- **Use** command-line tools (e.g. cURL) to **inspect** HTTP requests and responses for a given URL.
- **Identify** common HTTP headers (such as Content-Type, Set-Cookie, Cache-Control) and **outline** their roles in web communication.
- **Understand** how cookies enable session management and **discuss** basic security considerations for cookies.
- **Discuss** how web caching works and **recognise** the headers that control caching and content revalidation.
- **Explain** the concept of same-origin policy and **demonstrate** how Cross-Origin Resource Sharing (CORS) can allow controlled access to resources.
- **Construct** a well-structured HTML document with correct syntax, including all essential elements (doctype, head, body, etc.).
- **Utilise** semantic HTML elements (like <header>, <nav>, <section>, etc.) to organise content meaningfully.
- **Create** a simple HTML form to collect user input, using appropriate form elements and labels.
- **Apply** basic accessibility principles in HTML (such as alt attributes for images and labels for form fields) to make web content more inclusive.
- **Develop** a basic multi-section webpage (semantic homepage) as a hands-on exercise, demonstrating proper use of HTML and fundamental web concepts.

## 3. WEB FUNDAMENTALS

*Web Fundamentals* refer to the core concepts and technologies that enable the web. This includes the way web clients and servers interact, the protocol they use to communicate (HTTP), and related mechanisms like headers, cookies, caching, and security policies. In this section, we will break down these concepts to understand what happens behind the scenes each time you visit a website or web application.

### 3.1 Client-Server Model

The web is built on a **client-server model**. In this model, a **client** (usually a web browser on your computer or phone) sends a request for data or services, and a **server** (a remote computer hosting a website or API) responds with the requested data. The client and server

communicate over a network (the internet) using standard protocols. The client initiates the conversation, and the server serves the client's requests.

In practical terms, when you type a URL into your browser (the client), it will send a request to the server identified by that URL's domain name. The server, which is always listening for incoming requests, processes the request and sends back a response (often an HTML page, images, or data in another format). For example, if you visit **https://example.com**, your browser (client) sends a request to *example.com*'s server, and the server replies with the homepage HTML, which your browser then displays.

This architecture allows many clients to interact with one server at the same time. Servers are often powerful machines or cloud services designed to handle numerous requests concurrently. Clients are typically end-user devices running web browsers or other applications. Because the roles are distinct, the client-server model makes it easier to distribute workloads: the client handles the user interface and local tasks, while the server handles data storage, processing, and business logic.

**Exercise:** Identify three examples of client-server interactions in everyday internet use. For each example, name the client and the server. (For instance, when you use a mobile app to check the weather, the app is the client and the weather service's backend is the server.)

## 3.2 HTTP Basics

When a client and server communicate on the web, they follow the **HTTP protocol**. **HTTP** stands for *HyperText Transfer Protocol*. It is the foundational protocol of the web, defining how requests and responses are formatted and exchanged. HTTP is text-based and stateless, meaning each request from a client is independent of previous requests – the server does not automatically remember previous interactions without extra help (like cookies, which we will discuss later).

An HTTP **request** typically consists of:

- A **request line** (e.g. `GET /index.html HTTP/1.1`), which includes the HTTP *method* (such as GET or POST), the *path* or resource being requested, and the HTTP version.
- Request **headers**, which are key-value pairs providing additional information about the request (for example, what types of content the client accepts).
- An empty line, followed by an optional **body** (used especially in POST/PUT requests to send data).

The server answers with an HTTP **response** which has:

- A **status line** (e.g. `HTTP/1.1 200 OK`), which includes the HTTP version, a numeric *status code*, and a reason phrase. Status codes like 200 (OK), 404 (Not Found), 500 (Internal Server Error) indicate whether the request was successful or if an error occurred.

- Response **headers**, providing metadata about the response (such as content type and length).
- An empty line, followed by an optional **body** containing the requested content (HTML, JSON, image data, etc).

**Example:** Below is a simplified example of an HTTP GET request from a client and the server's response. This shows the request line, a few headers, and the start of the response body:

```
GET /index.html HTTP/1.1
Host: example.com
User-Agent: MyBrowser/1.0
Accept: text/html

HTTP/1.1 200 OK
Date: Tue, 05 Oct 2025 18:00:00 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 1256

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Example Domain</title>
  ...
```

In this example, the client requests the */index.html* page from **example.com** using HTTP/1.1. The server responds with a status code **200 OK**, indicating success, and includes headers like **Content-Type** (telling the browser that the content is HTML text). After the headers, the HTML content of the page is sent.

**Attention:** HTTP by itself does not encrypt data. This means information (including passwords or personal data) sent over plain `http://` can be intercepted. Always use **HTTPS** (HTTP Secure) for sensitive communications. HTTPS (accessed via `https://`) uses encryption (TLS) to protect data in transit, ensuring privacy and integrity.

HTTP defines several **methods**. The most common are **GET** (to retrieve data) and **POST** (to submit data). Other methods include **PUT**, **DELETE**, **HEAD**, etc., each intended for specific kinds of actions. Likewise, **status codes** are categorized: 100–199 for informational, 200–299 for success (200 OK, 204 No Content, etc.), 300–399 for redirection (301 Moved Permanently, 302 Found), 400–499 for client errors (400 Bad Request, the famous 404 Not Found), 500–599 for server errors (500 Internal Server Error, etc.).

Because HTTP is stateless, the server does not keep track of users between separate requests. For example, if you log in to a website and then click a link, the new request doesn't automatically "remember" you—unless something like a **cookie** is used, which we will cover soon.

**Exercise:** Why is HTTP described as "stateless"? Consider how a web application might *remember* that you are logged in between page visits if the protocol itself does not retain state. (*Hint:* Think about small pieces of data that the browser can send with each request to help the server recognize you, which will be discussed next.)

### 3.3 HTTP Headers

HTTP **headers** are an essential part of requests and responses. They provide metadata and control information that isn't part of the main content. Headers are sent as *key: value* pairs, one per line, immediately after the request or status line. Both clients and servers use headers to communicate details about the request/response or preferences.

Some common **request headers** include:

- **Host:** Specifies the hostname of the server (for virtual hosting). For example, `Host: example.com` tells the server which domain the client is trying to reach.
- **User-Agent:** Identifies the client software making the request (e.g., browser type/version). For example, a Chrome browser might send a User-Agent string identifying itself.
- **Accept:** Informs the server what content types the client can process (e.g. `Accept: text/html` or `Accept: application/json`).
- **Accept-Language:** Indicates the preferred languages (e.g. `Accept-Language: en-GB`, `en` might indicate a preference for British English if available).
- **Authorization:** Carries credentials for authentication (e.g. a token or username/password when accessing protected resources).

Typical **response headers** include:

- **Content-Type:** Describes the MIME type of the content in the response (e.g. `Content-Type: text/html` for an HTML page, or `application/json` for JSON data).
- **Content-Length:** The size of the response body in bytes.
- **Set-Cookie:** Instructs the browser to store a cookie (covered in the next section).
- **Cache-Control:** Instructions for caching (covered later).
- **Server:** Information about the server software (e.g. `Server: NGIX/2.4.1`).

Headers can convey a wide range of information. For example, the **Location** header (in a 3xx redirect response) tells the client where to go next, and **Content-Disposition** can suggest a file download name for attachments.

**Example:** Using a command-line tool to fetch headers. The `curl` tool is useful for fetching URLs. The `-I` option makes a HEAD request (fetching only headers). For example:

```
$ curl -I http://example.com
HTTP/1.1 200 OK
Date: Tue, 05 Oct 2025 18:05:00 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 1256
```

In this example, the tool requested the headers of *http://example.com*. The server's response (status 200 OK) shows a **Date** header (when the response was sent), **Content-Type** and **Content-Length**. Using such tools or browser developer tools, you can inspect headers to debug or understand how a web page is delivered.

**Exercise:** Pick a website and inspect the HTTP headers it sends. You can do this by using your web browser's developer tools (Network tab) or a tool like `curl`. Identify at least two request headers your browser sends, and two response headers the server returns. What does each of those headers do? (For instance, look for headers like **User-Agent**, **Content-Type**, or **Server** and try to explain their purpose.)

### 3.4 Cookies and Sessions

Since HTTP is stateless, web developers needed a way to **remember** information about a user across multiple requests – for example, to keep you logged in or to maintain items in your shopping cart. **Cookies** were introduced to achieve this. A cookie is a small piece of data that a server can ask the client to store, and the client will send it back with subsequent requests to the same server.

How cookies work:

- The server sends a **Set-Cookie** header in an HTTP response, instructing the browser to store a cookie. This header includes a name, value, and optional attributes (such as expiration, path, domain, security flags).
- The browser stores the cookie (usually as a small text file or in memory).
- On subsequent requests to the *same domain*, the browser automatically includes a **Cookie** header with the name and value of each cookie that the server set, so the server can recognize the client or recall stored information.

For example, when you log in to a website, the server might respond with a cookie like `Set-Cookie: sessionId=abc123; Path=/; HttpOnly`. The browser will store `sessionId=abc123`. On the next request to that site, the browser adds `Cookie: sessionId=abc123` to the request headers, so the server knows you're the same user with session ID `abc123`.

**Example:** Server setting and client returning a cookie via headers:

```
Server Response Header:
Set-Cookie: sessionId=abc123; Path=/; HttpOnly

Later Client Request Header:
Cookie: sessionId=abc123
```

In this example, the server's response told the browser to store a cookie named *sessionId* with value *abc123*. The *Path=/* attribute means the cookie is sent for all pages on the site. *HttpOnly* means the cookie is not accessible via JavaScript (for security). On the client's next request to the server, it automatically includes *sessionId=abc123* in the Cookie header.

Cookies enable **sessions** – the server can create a session record (like “user 42 is logged in”) and link it to a unique session ID stored in a cookie. Because the browser returns that session ID each time, the server knows who you are and that you're logged in. Cookies can also store preferences (e.g., preferred site language) or tracking data.

**Attention:** Cookies often contain sensitive information (or identifiers to sensitive server-side state). It's important to secure them. Cookies should be marked **Secure** (only sent over HTTPS) if they are used in an HTTPS site, and **HttpOnly** if they should not be accessible via JavaScript (to mitigate certain attacks). Also, be mindful of privacy: cookies can be used to track user behaviour across sites (especially via third-party cookies).

**Exercise:** Check the cookies stored in your browser for one of your frequently used websites. Most browsers allow you to view cookies in their settings or developer tools. What are some names of cookies you see, and can you guess their purpose (e.g., session identifier, preferences, tracking)? Consider which might be session cookies (temporary) vs. persistent cookies (with an expiration date).

### 3.5 Caching on the Web

Web **caching** is a technique to improve performance and reduce unnecessary network traffic. The idea is that if a resource (like an image, stylesheet, or even an entire HTML page) has been fetched once, it might not need to be fetched again from the server every time the user revisits the page or another page that uses that resource. Instead, the browser (or an intermediary cache) can store a copy and reuse it, as long as it's considered fresh.

HTTP provides headers to control caching behavior:

- **Cache-Control:** A key header that specifies caching policies. For example, *Cache-Control: max-age=3600* means “this response can be cached and is valid for the next 3600 seconds (1 hour).” During that time, the browser will use the cached copy without re-fetching from the server.
- **Expires:** An older header (largely superseded by Cache-Control) that gives an absolute date/time after which the response is considered stale.
- **ETag:** An identifier for a specific version of a resource (e.g., a hash of the content). The server might send *ETag: "abc123"* with a resource.
- **Last-Modified:** The date/time the resource was last changed (e.g., *Last-Modified: Tue, 05 Oct 2021 15:00:00 GMT*).

When a cached resource becomes stale or the browser wants to check for updates, it can make a **conditional request** using headers:

- **If-None-Match:** with the ETag value (e.g., If-None-Match: "abc123"), asking “send me the resource only if it doesn’t match this ETag.”
- **If-Modified-Since:** with a date (e.g., If-Modified-Since: Tue, 05 Oct 2021 15:00:00 GMT), asking “send me the resource only if it has been modified after this date.”

If the resource has not changed, instead of resending the whole thing, the server can reply with **HTTP 304 Not Modified**, which has no body. This tells the browser “your cached copy is still good.” This saves bandwidth and time.

**Example:** A caching scenario with headers:

```
HTTP/1.1 200 OK
Content-Type: image/png
Last-Modified: Tue, 05 Oct 2021 15:00:00 GMT
Cache-Control: max-age=86400

... [image data] ...

-- Later Request -->
If-Modified-Since: Tue, 05 Oct 2021 15:00:00 GMT

<-- Server Response --
HTTP/1.1 304 Not Modified
```

In this flow, the server initially sent an image with a **Last-Modified** timestamp and told the client it can cache it for 86400 seconds (a day) via **Cache-Control**. If the client checks the next day, it includes If-Modified-Since with the date it previously saw. The server responds 304 Not Modified if the image hasn’t changed, telling the browser to use its cached copy (and not sending the image data again).

Caching greatly speeds up web browsing: the second time you visit a page (or when you navigate within a site), many resources can load from cache almost instantly. Developers can control caching to ensure users get updates when needed (by changing file names or ETags when content changes, or setting appropriate max-age and must-revalidate directives).

**Note:** When developing or troubleshooting, you might need to **clear cache** or do a "hard refresh" (which forces the browser to re-download everything) to see changes. But for live sites, effective use of caching is essential for performance.

**Exercise:** Open the developer tools in your web browser and load a page twice in a row. On the second load, inspect the network log. Do you see some resources marked as "Cached" or "304 Not Modified"? Identify one such resource and note its caching headers. This will give you a concrete example of caching in action on a real site.

## 3.6 Cross-Origin Resource Sharing (CORS)

Web browsers enforce a security mechanism called the **same-origin policy**. In simple terms, this policy restricts web pages from making requests to a different domain than the one that served the web page, especially if the request is attempting to read data. For example, a script on `http://siteA.com` can't freely fetch data from `http://siteB.com` and read the response due to same-origin policy. This is to prevent malicious sites from stealing data from other sites by exploiting a user's logged-in status.

However, there are legitimate scenarios where cross-site requests are needed – for instance, a web application might need to fetch data from an external API. This is where **Cross-Origin Resource Sharing (CORS)** comes in. CORS is a mechanism that allows a server to specify that certain cross-origin requests are permitted.

**How CORS works:** When a browser makes a cross-origin request (using JavaScript, e.g. via fetch or an AJAX call), it will look for specific response headers from the target server:

- **Access-Control-Allow-Origin:** This header indicates which origins are allowed to receive the response. For example, a server might send `Access-Control-Allow-Origin: https://siteA.com` to explicitly allow siteA to use the response. It could also use `*` as a wildcard to allow any site (for public APIs, for instance).
- **Access-Control-Allow-Methods:** On certain requests (particularly non-simple requests or preflight responses), the server lists which HTTP methods are permitted for cross-origin requests (GET, POST, etc.).
- **Access-Control-Allow-Headers:** If the request includes special headers, the server must explicitly allow them by listing them in this header.
- **Access-Control-Allow-Credentials:** If the request includes credentials (cookies or HTTP auth) and the server is willing to allow them, this header must be set to `true` in the response.

For "non-simple" requests (e.g., those with custom headers or methods beyond GET/POST), the browser will send a **preflight** request (an HTTP OPTIONS request) to the server first, asking if the actual request is allowed. The server must respond with the appropriate CORS headers to approve the real request.

If the CORS headers indicate the request is not allowed, the browser will block the response data from being accessible to the web page (even though the request might have technically succeeded). Essentially, without proper CORS permission, you might see an error in the browser console like "Cross-Origin request blocked".

**Example:** A cross-origin request scenario:

- A script on `https://siteA.com` tries to fetch data from `https://api.siteB.com/data`.
- The browser adds an `Origin: https://siteA.com` header to the request to

tell siteB where the request is coming from.

- If siteB's server is configured to allow this, its response might include:

Access-Control-Allow-Origin: https://siteA.com

- If that header matches the requesting origin, the browser allows siteA's script to access the response. If siteB instead responded with no CORS header or a non-matching origin, the browser would block siteA's script from using the response.

**Note:** CORS restrictions are enforced by browsers. Other tools like curl or server-side code can fetch from any domain without these same-origin limitations. The same-origin policy (and thus CORS) is a *browser security* feature.

**Exercise:** Suppose your web application at **https://mySite.com** needs to fetch data from an API at **https://dataProvider.com**. You encounter a CORS error in the browser. What steps would you take to resolve it? (*Hint: Consider what the API server at dataProvider.com would need to send in its headers to let your domain access the data. How could you configure or request the API to enable CORS for your site?*)

### 3.7 LAB: Inspecting HTTP with Command-Line Tools

In this lab, you will use command-line tools to send HTTP requests and observe the responses. This will deepen your understanding of the HTTP traffic that your browser normally handles behind the scenes.

**Goal:** Use the curl command-line tool (or a similar utility) to perform an HTTP request and observe the full request and response, including headers and body.

#### Steps:

1. **Open a terminal (command prompt):** Ensure you have access to the curl tool. (On many systems it's pre-installed. On Windows, you can use PowerShell's built-in curl alias or install curl separately. Modern Windows 10/11 also include curl by default.)
2. **Make a basic request:**  
Run the command:

```
curl -v http://example.com
```

The -v option stands for "verbose". This tells curl to output detailed information about the request and response.

3. **Observe the output:** The curl output will display:
4. Lines prefixed with > — these are the request headers sent by curl. For example, you should see something like:  

```
> GET / HTTP/1.1  
> Host: example.com
```

```
> User-Agent: curl/7.xx  
> Accept: */*
```

This shows the request method (GET), the path (/), and some default headers curl sends (User-Agent and Accept).

5. Lines prefixed with < — these are the response headers from the server. You might see:

```
< HTTP/1.1 200 OK  
< Content-Type: text/html; charset=UTF-8  
< Content-Length: ...  
< Date: Tue, 05 Oct 2025 18:10:00 GMT
```

followed by a blank line and then the HTML content (which may start with <!DOCTYPE html>).

6. The HTML of the page will likely be printed after the headers. This is the raw HTML source code of example.com's homepage.

A truncated example of what part of the output might look like is:

```
> GET / HTTP/1.1  
> Host: example.com  
> User-Agent: curl/7.68.0  
> Accept: */*  
< HTTP/1.1 200 OK  
< Content-Type: text/html; charset=UTF-8  
< Content-Length: 1256  
< Date: Tue, 05 Oct 2025 18:10:00 GMT  
  
<!DOCTYPE html>  
<html>  
<head><title>Example Domain</title></head>  
<body> ... (HTML content)...
```

1. **Analyze the response:** Note the status code in the response (should be 200 for example.com). Identify the response headers like Content-Type and any others present. The body is the HTML that the browser would render. In this raw form, it's not very pretty to read, but this is exactly what your browser receives before it renders the page.
2. **Fetch headers only (optional):** If you want to see just the headers without the body, try:

```
curl -I http://example.com
```

This sends a HEAD request and should output only the response headers (as we saw in an earlier example).

3. **Try another site or page:** If you have time, try using curl on another URL. For instance:

```
curl -v http://example.com/robots.txt
```

This will fetch the robots.txt file. Notice that the output might be short (as it's a small text file) and you'll still see headers. You could also try a site that redirects (like <http://github.com>, which redirects to <https://github.com>) to observe a 301/302 redirect and the Location header.

4. **Close the terminal:** When done, you can close the terminal. There is no persistent change made by curl – it's just retrieving data.

**What you learned in this lab:** You manually performed web requests and saw exactly what is exchanged between client and server. This demystifies what the browser is doing when it loads pages. You saw the request line and headers (and how tools like curl format/show them), as well as the server's reply including headers and body. Being comfortable with inspecting HTTP in this way is very useful for debugging web applications.

*(If you cannot use curl, an alternative is to use an online HTTP request inspector or a browser extension for viewing HTTP requests. However, curl is a versatile tool worth getting familiar with.)*

## 4. HTML Essentials

Having explored the communication side of the web, we now turn to the content side: **HTML (HyperText Markup Language)**. HTML is the standard language used to create web pages. It provides the structure and meaning of content, which web browsers interpret and display to users. In this section, we cover the essentials of HTML: its syntax and structure, the use of semantic elements to write meaningful markup, forms for user input, and accessibility considerations to ensure our content can be used by everyone.

### 4.1 HTML Syntax and Structure

HTML is a *markup language*, which means it uses **tags** enclosed in angle brackets (e.g., `<tag>`) to annotate content. Each pair of tags defines an **element**. Most HTML elements have an opening tag (like `<p>`) and a closing tag (like `</p>`) that wrap content; the closing tag has a forward slash to indicate the end. Some elements are self-closing (void elements) and don't wrap content (e.g., `<img>` or `<br>`).

An HTML document has a defined **structure**. A basic HTML5 document looks like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My First Page</title>
```

```
</head>
<body>
  <h1>Hello, world!</h1>
  <p>This is a simple web page.</p>
</body>
</html>
```

Let's break down this structure:

- The `<!DOCTYPE html>` declaration at the very top tells the browser that this is an HTML5 document. It's not a tag per se, but an instruction for the browser's rendering engine.
- The `<html>` element wraps all the content of the page. The `lang="en"` attribute declares the language (English in this case) for accessibility and search engines.
- Inside `<html>`, there are two main sections:
- `<head>`: This section contains metadata about the page (not displayed as page content).

In the example, `<meta charset="UTF-8">` declares the character encoding (UTF-8 is standard), and `<title>` defines the text that appears in the browser tab or window title for the page.

- `<body>`: This section contains the actual content of the page that will be displayed to the user. In the example, the body has an `<h1>` heading and a `<p>` (paragraph) element.
- Each tag should be properly opened and closed, and elements should be nested in a logical, non-overlapping way. Proper nesting is important (e.g., you shouldn't close a parent element before closing its child elements).
- HTML is *not* case-sensitive (you could write `<HTML>` or `<html>`), but the convention is to use lowercase for all tags and attributes.

Beyond the essentials shown above, HTML provides a wide range of elements to mark up content:

- Headings: `<h1>` through `<h6>` for titles and section headings (h1 is the most important, h6 the least).
- Paragraphs: `<p>` for blocks of text.
- Links: `<a href="URL">` for hyperlinks (with the `href` attribute specifying the destination).
- Images: `` to embed images (with `src` for the image path and `alt` for alternate text).
- Lists: `<ul>` (unordered list, bullet points) and `<ol>` (ordered list, numbered) containing `<li>` list items.
- *And many more*: tables (`<table>`), forms, video/audio embeds, etc., though those are beyond the basics for now.

**Exercise:** Using a plain text editor, create an HTML file called **hello.html**. In that file, type the basic structure as shown above: include `<!DOCTYPE html>`, an `<html>` element with a `<head>` (containing at least `<meta charset="UTF-8">` and a `<title>`), and a `<body>` with some content. For example, put a heading with your name and a paragraph introducing yourself. Save the file and open it in a web browser (you can just double-click it or drag it into a browser window). If everything is correct, you should see your heading and paragraph displayed. This exercise will give you hands-on experience with writing and viewing a simple HTML page.

## 4.2 Semantic HTML Elements

As HTML has evolved (especially with HTML5), a key focus has been on **semantic** elements – tags that convey the meaning or role of content, not just how it looks. Using semantic HTML improves the clarity of your markup for both developers and user agents (like browsers, search engines, screen readers).

Some important semantic elements introduced in HTML5 include:

- `<header>` – Defines a header for a page or section (often contains the site title, logo, navigation links, etc.).
- `<nav>` – Defines a block of navigation links.
- `<main>` – Represents the main content of a document. There should typically be one `<main>` per page, containing the primary content (excluding headers, footers, nav, etc.).
- `<section>` – Defines a thematic grouping of content, typically with a heading. Sections are used to break content into logical chunks.
- `<article>` – Represents a self-contained composition that could stand on its own (like a blog post, news article, forum post, etc.). It often contains its own header, footer, etc., and could potentially be syndicated or reused.
- `<aside>` – Represents a portion of the page with content indirectly related to the main content (like a sidebar, pull quote, list of related links, advertisements, etc.).
- `<footer>` – Defines a footer for a page or section (often contains information like author, copyright, related links, or at the very bottom of the page, site credits or contact info).
- `<figure>` and `<figcaption>` – Used for self-contained content like images or diagrams, along with a caption. `<figure>` can wrap an image and its `<figcaption>` to semantically tie a caption to the image.

Using these elements in place of generic containers (like the all-purpose `<div>` tag) makes the structure more meaningful. For example, instead of many `<div id="header">` or `<div class="nav">` containers, we now have dedicated tags `<header>` and `<nav>` that instantly communicate their purpose.

**Example:** A structured page layout with semantic elements:

```

<body>
  <header>
    <h1>My Website</h1>
    <p>Welcome to my homepage!</p>
  </header>

  <nav>
    <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="about.html">About Me</a></li>
      <li><a href="contact.html">Contact</a></li>
    </ul>
  </nav>

  <main>
    <article>
      <h2>About Me</h2>
      <p>Hello! My name is Alice, and this is my personal website. I am
learning web development and excited to share my journey.</p>
    </article>

    <aside>
      <h3>Latest News</h3>
      <p>I recently started a blog about tech and art. Check out the <a
href="blog.html">latest posts</a>!</p>
    </aside>
  </main>

  <footer>
    <p>&copy; 2025 Alice's Website</p>
  </footer>
</body>

```

In this example:

- The <header> contains the main heading of the site (and could include a tagline or intro text).
- The <nav> contains a list of links for site navigation.
- The <main> encapsulates the central content. Inside it, there's an <article> (with a section of content "About Me") and an <aside> for side content (perhaps recent news or additional info).
- The <footer> closes out the page with a copyright notice.

Notice how each part of the page is marked with a tag that describes its role. This not only makes the HTML easier to understand at a glance, but also helps search engines and accessibility tools. For example, a screen reader can allow a user to jump directly to the navigation or main content because those regions are labeled with <nav> and <main> tags.

Using semantic tags doesn't change the default appearance much (that's what CSS is for), but it lays a strong foundation for a well-organised page. It also often improves SEO (search engines know, for example, what text is in headings or navigation menus) and is crucial for accessibility.

**Note:** Don't overuse semantic tags; use them when they make sense. For instance, not every `<div>` needs to be a `<section>` or `<article>` – only when the content is logically a separate section or a standalone piece. It's fine to still use `<div>` or `<span>` for generic containers or styling hooks when no specific semantic element fits.

**Exercise:** Think of a simple webpage idea (for example, a personal profile or a small business homepage). Outline on paper (or in a text document) the different sections of content it would have. Then assign HTML5 semantic elements to each part of your outline. For instance, identify what would be in the header, what might form an article or section, and what could go in the footer. This planning exercise will help you structure content semantically before even writing the HTML code.

## 4.3 HTML Forms

HTML **forms** are the primary way to collect input from users on the web – whether it's a sign-up form, a search box, or a contact form. A form is defined by the `<form>` element and usually contains various input elements where the user can enter data, like text fields, checkboxes, radio buttons, drop-down lists, and buttons.

Key aspects of forms:

- The `<form>` tag often includes:

- **action** attribute: the URL to which the form data will be sent when the form is submitted.

- **method** attribute: the HTTP method to use (typically "GET" or "POST"). GET will append the form data to the URL (suitable for non-sensitive data or searches), whereas POST will send the data in the request body (better for larger amounts of data or sensitive info, because it doesn't get displayed in the URL).

- **Input elements:**

- `<input>` is a versatile tag for many types of inputs. It has a **type** attribute that defines what kind of input (text, email, password, checkbox, radio, button, etc.), as well as **name** (which labels the data when submitted), **id** (to link with labels or for scripting), **value** (for an initial value or, in the case of buttons, the button label), and more.

- Other form elements include `<textarea>` for multi-line text input, `<select>` for drop-down lists (with nested `<option>` tags for each choice), and `<button>` (which can be an alternative to `<input type="submit">` for a

submit button, or type="button" for a clickable button that doesn't submit the form by itself).

- **Label elements:** <label> is used to provide a text label for an input. The label has a for attribute that should match the id of the corresponding input. This association not only visually labels the field but also improves accessibility (screen readers will read the label when the field is focused, and clicking the label focuses the field).

Let's look at an example form:

```
<form action="/signup" method="POST">
  <label for="username">Username: </label>
  <input type="text" id="username" name="username" required>

  <br>

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

  <br>

  <label for="age">Age: </label>
  <input type="number" id="age" name="age">

  <br>

  <button type="submit">Sign Up</button>
</form>
```

In this form example:

- The form is set to submit via POST to the "/signup" URL on the server.
- It has three input fields: - a text field for "Username,"
- an email field for "Email,"
- a number field for "Age."

Each has a corresponding <label> with a for attribute linking to the input's id. This makes it so that clicking the label focuses the field, and it helps screen readers by announcing the label when the input is focused.

- The required attribute on the username and email fields means the browser will require those fields to be filled in before allowing submission (basic HTML5 validation).
- A <button type="submit"> is provided to submit the form. (An equivalent in older practice is <input type="submit" value="Sign Up">, which achieves the same result.)
- <br> tags are used here to insert line breaks purely for layout in this raw

example. In a real form, you might use CSS for layout instead of multiple `<br>` tags.

When the user fills out this form and clicks “Sign Up,” the browser will send a POST request to “/signup.” The form data will be encoded (by default as application/x-www-form-urlencoded) and sent in the request body. For example, it might send something like `username=Alice&email=alice%40example.com&age=30` (with special characters like “@” URL-encoded as “%40”).

If the method were “GET”, the same data would be appended to the URL after a `?` (for instance: `/signup?username=Alice&email=alice%40example.com&age=30`), which is one reason not to use GET for sensitive data like passwords – the info becomes part of the URL (visible in browser history and server logs).

**Attention:** Avoid using GET for forms that submit sensitive information (like passwords or personal details) because these will appear in the URL and could be logged or cached. Use POST for such cases, which keeps the data in the request body (and not in the URL).

Forms are the cornerstone of interactive web pages – enabling user registration, searches, feedback submission, and more. On the server side, there needs to be a script or application logic to handle the form submission (the details of server-side processing are beyond our scope here, but it’s good to remember that after submission, the server will parse the incoming data and respond accordingly).

**Exercise:** Create a simple HTML file called **feedback.html** that contains a feedback form. Include the following fields: Name (text), Email (email), and Comments (textarea for multi-line input). Add a submit button. You don’t need an actual server script – for now, you can set the form’s action to something like `/feedback` and method to “GET” just to test (or simply leave it and the browser will attempt to request a page, which will likely 404). Open the file in your browser and try filling it out and submitting – you will see the form data appear in the URL (with GET) or if using POST you might just see a blank page after submission because there’s no server handling it. The purpose is to practice writing form elements and linking labels to inputs.

## 4.4 Accessibility Basics

Web **accessibility** means designing and developing websites so that people with disabilities can use them. This includes users who may be blind or visually impaired, deaf or hard of hearing, motor impaired (difficulty using a mouse), or cognitively impaired. Many accessibility best practices align with using semantic, well-structured HTML – which we’ve already touched on – and ensuring alternatives are provided for non-text content.

Basic HTML accessibility guidelines:

- **Use proper structure and semantics:** Headings should be actual `<h1> . . . <h6>` elements in a logical order (this helps screen reader users navigate the page by heading). Lists should be marked up with `<ul>/<ol>` for list content. Tables should

use proper table tags (<table>, <tr>, <th>, <td>) with headings where appropriate. Avoid using elements for something they're not (e.g., don't use a <table> for page layout, or a <p> just to style as a heading). Proper structure ensures assistive technologies can understand the page hierarchy and relationships.

- **Provide text alternatives:** For any non-text content (images, audio, video):

- Images should have an **alt attribute** (). The alt text describes the image for those who cannot see it. If an image is purely decorative, alt can be left empty (alt="") to indicate it can be ignored by screen readers.

- Videos should ideally have captions or a text transcript for audio content. Audio-only content should have a transcript or summary available.

- **Form labels:** Every form input should have a visible text label, connected with a <label for="..."> as shown in the forms section. This ensures that screen readers announce the label when the field is focused. It also increases the clickable area (clicking the label focuses the field).

- **Keyboard navigation:** Ensure that all interactive elements (links, buttons, form fields, etc.) can be accessed via keyboard alone. Typically, if you use standard HTML controls, this is taken care of. For example, links and form controls are naturally focusable and operable by keyboard (using Tab, Enter, Space, etc.). If you create custom interactive elements (like a <div> that acts like a button), you must add proper attributes (like tabindex="0") and event handlers to make them focusable and triggerable via keyboard. It's much easier to use real <button> and <a> elements, which have built-in keyboard support.

- **ARIA (Accessible Rich Internet Applications) attributes:** These are special attributes (like role, aria-label, aria-hidden, etc.) that can enhance accessibility when plain HTML isn't enough. For example, aria-label can provide an accessible name for an element that doesn't have visible text. However, ARIA should be used sparingly and appropriately – the mantra is *"Use native HTML elements where possible, and use ARIA only when needed."* If you stick to semantic HTML, you often don't need to add much ARIA.

- **Color and contrast:** Ensure that text has sufficient contrast against its background so that it can be read by people with low vision or color blindness. For example, light gray text on white is hard for many to read. There are standard contrast ratios defined in accessibility guidelines (WCAG). Although choosing colors is more related to CSS design, as a content creator you should be mindful not to rely on color alone to convey information (e.g., don't say "important items are in red" without also providing a textual cue, because a color-blind user might not distinguish red).

- **Skip links:** For long pages, providing a “Skip to main content” link at the top allows keyboard users to jump past repetitive navigation directly to the main content. This link is often hidden visually (until focused) but is accessible to screen readers or when tabbing.

**Example:** Accessible markup improvements:

```
<!-- Image with alt text -->


<!-- Form input with proper label -->
<label for="phone">Phone number:</label>
<input type="tel" id="phone" name="phone" placeholder="123-456-7890">
```

In the first line, the image has alternate text so if the image doesn’t load or for a screen reader user, “Company Logo” will be conveyed. In the second snippet, the input for phone has a label. The placeholder provides an example format but is not a substitute for a clear label (placeholders disappear when you start typing and should not be solely relied upon for identification).

**Attention:** It’s not just about ticking boxes – accessible websites benefit *everyone*. They are often better structured, easier to maintain, and SEO-friendly. More importantly, failing to consider accessibility can exclude a significant portion of users and may even have legal implications in many regions. Always consider accessibility from the start of a project, not as an afterthought.

**Exercise:** Try navigating your own webpage (or any website you frequent) using only the keyboard. Use the Tab key to jump through links and form fields, and use Enter to activate links or buttons. Can you access all features? Is the focus indicator visible as you tab through elements? This simulates how someone who cannot use a mouse would navigate. Next, if you have a screen reader available (for instance, Narrator on Windows or VoiceOver on Mac), turn it on and listen to how it reads a page. Does it convey the content in a sensible order? Are images described (via alt text) and form fields announced properly? This exercise can reveal a lot about the accessibility of a page.

## 4.5 Lab: Building a Semantic Homepage

In this lab, you will apply what you’ve learned about HTML structure, semantic elements, forms, and accessibility to build a simple homepage. This will be a hands-on exercise to create an HTML page that could serve as a personal or project homepage, using semantic HTML throughout.

**Scenario:** Imagine you are creating a personal homepage. It should include a header, a navigation menu, a main content area with at least one article or section of text, an aside for secondary information, and a footer. You can decide the content (for example, it could be an “About Me” page, or a homepage for a fictional small business or portfolio). Focus on using semantic elements for each part.

## Tasks:

1. **Set up the file:** Create a new file called `index.html`. Start by typing the basic HTML5 document structure: include the doctype, `<html>` with a language attribute, a `<head>` with `<meta charset="UTF-8">` and a relevant `<title>`, and an empty `<body>`.
2. **Header:** Inside the `<body>`, add a `<header>` element. In the header, include:
  - An `<h1>` with the site or your name (e.g., “Alice Doe” or “Alice’s Portfolio”).
  - Possibly a short tagline or welcome message in a paragraph or smaller heading.
  - *(Optional)* If you have a logo image, you could include an `<img>` inside the header with an appropriate `alt` attribute.
3. **Navigation:** Either within the header or right below it, include a `<nav>` element for your navigation menu. Inside, create an unordered list (`<ul>`) of links (`<li><a href="...">Link text</a></li>` for each). For example: Home, About, Contact (even if those other pages don’t exist yet, you can link to “#” or to actual filenames if you plan to create them). This will serve as your site menu.
4. **Main content:** Add a `<main>` element to contain the primary content of the page. Within `<main>`:
  - Create an `<article>` (or you can use a `<section>` if more appropriate) that contains a standalone piece of content. For a personal homepage, this might be an “About Me” section. Use an `<h2>` for this section title and follow it with one or more `<p>` paragraphs with information about you or the site.
  - Optionally, if you have more content, you could include additional sections or articles (for instance, a section for “Projects” or “Blog Highlights”, each with an `<h2>` and some content). For now, one article/section is sufficient.
5. **Aside (sidebar):** Include an `<aside>` element for secondary content. This could be within the main content area (if it’s related to the main content) or after the main content. For example, on a personal site, the aside could contain a list of recent posts, a short bio blurb, or external links. Give it an `<h3>` if it needs a title (like “Latest News” or “Quick Links”), and content such as a list of links or a small paragraph.
6. **Footer:** Add a `<footer>` at the bottom of the body. In the footer, include some or all of:
  - A small copyright notice, e.g., `&copy; 2025 Your Name.`

- A contact email or link to a contact page.
- This is also a good place for any legal notices or a link to a privacy policy if this were a real site.

7. **Review and accessibility:** Go through your HTML and ensure:

- Every image has alt text (if any images are included).
- Every form control (if you added any, perhaps a newsletter sign-up) has a label.
- The heading levels make sense (e.g., one `<h1>` for the site title, `<h2>` for section headings, etc., without skipping levels).
- If possible, open the page in a browser and try navigating with Tab to ensure you can reach the nav links and any other interactive elements.

8. **Save and view:** Open your `index.html` in a web browser. You should see your structured content. It won't be styled (everything will appear in default browser styling, e.g., headings bold and large, list with bullets, etc.), but focus on the structure being correct. Verify that the content is all there and structured semantically.

9. *(Optional bonus):* If you know a bit of CSS, you can add a simple `<style>` block in the head or link a CSS file to add basic styling to your page (for example, set a background color for the header, or style the nav links horizontally). This is not required for this lab, since the focus is on HTML, but feel free to experiment.

This lab brings together many concepts: you practiced creating the essential parts of an HTML page (head, body), used semantic elements like header, nav, main, article, aside, and footer to organise content, and considered basic accessibility by using proper tags. You now have a basic static webpage that could serve as a starting point for further expansion or styling in the next stages of web development.

## 4.6 Intermediate Exercises

Now that you have learned the fundamentals of web communication and HTML, here are some additional exercises to extend your knowledge and skills. These will help reinforce what you've learned and push you a bit further:

- **Validate and Debug:** Take the HTML page you built in the lab (or any HTML page you've created) and run it through an HTML validator (like the W3C Markup Validation Service). Fix any errors or warnings that come up. This will help you write cleaner, standards-compliant HTML. Also, try viewing your page in multiple browsers (Chrome, Firefox, etc.) to ensure it behaves consistently.
- **Extend Your Homepage:** Expand on the semantic homepage you created. For example, add another `<section>` or `<article>` to the main content for "Projects" or

“Blog Posts” and include a few sample items. Within that section, you could use a list (`<ul>` or `<ol>`) to list project titles or blog post links. Practice using appropriate tags for each piece of content (e.g., use the `<time>` tag to mark up dates, if any, with a `datetime` attribute).

- **Create a Second Page and Link It:** Make another HTML page (e.g., `about.html` or `contact.html`) with content appropriate to its purpose. Ensure it has proper structure (perhaps a smaller header or just a heading, some paragraphs, etc.). Then link this new page from your homepage’s nav menu (and add a link back to home on the new page). This will give you a mini website with navigation.
- **Add a Form:** On either your homepage or a new page, add a form for something like a newsletter sign-up or contact form. Use at least two different types of inputs (e.g., text and email, or text and textarea, etc.), and make sure to include labels for each. For extra practice, try using the `<fieldset>` and `<legend>` elements to group related form fields (for example, group a set of contact info fields under a fieldset with a “Contact Information” legend).
- **Accessibility Audit:** Choose a popular website (maybe a news site or a government site) and inspect how they structure their HTML for accessibility. Try using a screen reader or an online accessibility checker on that site and note how it performs. For instance, is it easy to navigate via headings? Do images have alt text? This will train you to think critically about accessibility in real-world sites.
- **Experiment with Media:** If you’re feeling adventurous, try adding an HTML5 video or audio element to a page. For example, use the `<video>` tag with `controls` and a `source` to embed a sample video (you can find free sample videos online, or use a short one you have). Ensure you provide either captions or a description for the video. Similarly, try an `<audio>` tag with a sample sound clip. Even if you don’t have your own media files, knowing how to embed them with HTML (e.g., `<video width="320" height="240" controls> <source src="sample.mp4" type="video/mp4"> </video>`) is useful for future projects.

Each of these intermediate exercises is designed to deepen your understanding. They cover validating code, expanding semantic structure, linking multiple pages (which introduces the concept of a website vs. a single page), handling user input with forms, considering accessibility on existing sites, and using HTML for multimedia. Tackle them at your own pace, and don’t hesitate to consult documentation (MDN Web Docs is an excellent resource) when trying new HTML elements or attributes.

## 5. What We Have Learned

- **Client-Server Architecture:** The web operates on a client-server model where web browsers (clients) send requests and web servers respond with data. Understanding this interaction is key to grasping how any web page loads.

- **HTTP Protocol Fundamentals:** We learned that HTTP is the foundational communication protocol of the web. It uses methods like GET and POST to request resources, and servers reply with status codes (200, 404, 500, etc.) and content. HTTP is stateless, meaning each request is independent unless we introduce mechanisms to maintain state.
- **Role of HTTP Headers:** Headers in HTTP requests and responses carry important information such as content type, caching instructions, cookies, and more. We saw examples of common headers and how to inspect them using tools like browser developer tools or curl. Headers can control behaviors (like redirects via Location, or caching via Cache-Control headers).
- **Cookies and Session Management:** Cookies provide a way to remember information about users between requests, enabling sessions (like keeping a user logged in or remembering preferences). We discussed how servers set cookies and how browsers send them back, as well as basic security practices for cookies (HttpOnly, Secure flags).
- **Caching for Performance:** Web caching mechanisms store copies of resources to make subsequent loads faster. We covered how servers use headers like Cache-Control, ETag, and Last-Modified to guide caching, and how browsers can revalidate resources (resulting in 304 Not Modified responses). Effective caching greatly improves user experience by reducing load times and saving bandwidth.
- **CORS and Browser Security:** The same-origin policy protects users by restricting cross-site requests. Cross-Origin Resource Sharing (CORS) is the controlled way to relax those restrictions. We learned that servers must explicitly allow cross-origin requests by sending appropriate headers, and that this is a concern primarily for front-end developers working with third-party APIs (since server-to-server requests aren't limited by same-origin rules).
- **HTML Document Structure:** We mastered the basic structure of an HTML5 document – using the correct doctype, <html>, <head>, and <body> tags, and including essential metadata like character encoding and page title. A well-structured HTML document is the foundation of any webpage.
- **Semantic HTML and Best Practices:** Using semantic elements (header, nav, main, section, article, aside, footer, etc.) makes our HTML more meaningful and easier to maintain. We've seen how to structure a page layout using these elements instead of generic <div>s, which improves clarity and accessibility.
- **Building Forms:** Forms in HTML allow user input to be collected and sent to a server. We learned to use the <form> tag with action and method, and various form controls like <input> (with types like text, email, number), <textarea>, <select>, and <button>. We also emphasized the importance of labels for accessibility.

Understanding forms lays the groundwork for creating interactive sites (though handling the data requires server-side code).

- **Accessibility Considerations:** We highlighted the importance of making web content accessible. This includes providing alt text for images, using proper labels and structural elements, ensuring keyboard navigability, and generally designing with inclusivity in mind. Accessibility is not only ethically and legally important, it often goes hand-in-hand with good semantic HTML and overall good design.
- **Hands-On Practice:** Through the lab exercises, you practiced inspecting network requests via the command line and creating a structured HTML page from scratch. These practical activities reinforced the theoretical knowledge and gave you confidence in applying what you learned.

In summary, we've covered both the *invisible* workings of the web (how data gets from server to browser, and how browsers handle that data) and the *visible* structure of web content (how to write HTML that structures content logically and accessibly). These fundamentals are an essential stepping stone towards more advanced web development topics like CSS (for presentation) and JavaScript (for interactivity).

## 6. Brief Recap

In this document, we started our journey into web development by exploring fundamental concepts that every web developer should understand. We looked at how the web functions at a basic level – demystifying the client-server communication that occurs every time you load a webpage – and we built a solid foundation in HTML, the language that structures the content of those webpages.

By learning about HTTP and practicing with tools to inspect network communications, you have gained insight into what happens behind the scenes when you browse the web. By writing HTML pages using semantic elements and good practices, you've taken the first steps in creating content for the web that is well-structured and accessible.

This is just the beginning. With the fundamentals of HTML in place, the next natural step is to make your pages more visually engaging. In the **next segment**, we will introduce **CSS (Cascading Style Sheets)**. CSS is the technology used to style and layout web pages – controlling layout, colors, fonts, and the overall look and feel of your site. Where HTML provides the structure and content, CSS provides the presentation. Together, they allow you to create attractive, user-friendly websites.

As you move forward, remember to continually apply the principles learned here: keep your HTML semantic and organized, test and understand the behavior of your pages (both in how they function and how they appear to different users or in different environments), and keep performance and accessibility in mind. These habits will serve you well not just in the next segment on CSS, but throughout your web development journey.