WebPage & deployment

Introdução Engenharia Informática

Mário Antunes

November 17, 2025

Universidade de Aveiro

Table of Contents i

What is a “Static” Webpage?

- A **static webpage** is a file (or set of files) delivered to a user’s web browser *exactly* as it is stored on the server.
- The content is **fixed**. Every user sees the exact same content, regardless of who they are or what they do.
- Think of it as a digital brochure or a page in a book.
- The server’s job is simple: find the file (e.g., `about.html`) and send it.
- Core technologies: **HTML** (for structure) and **CSS** (for style).

What is a “Static” Webpage? ii

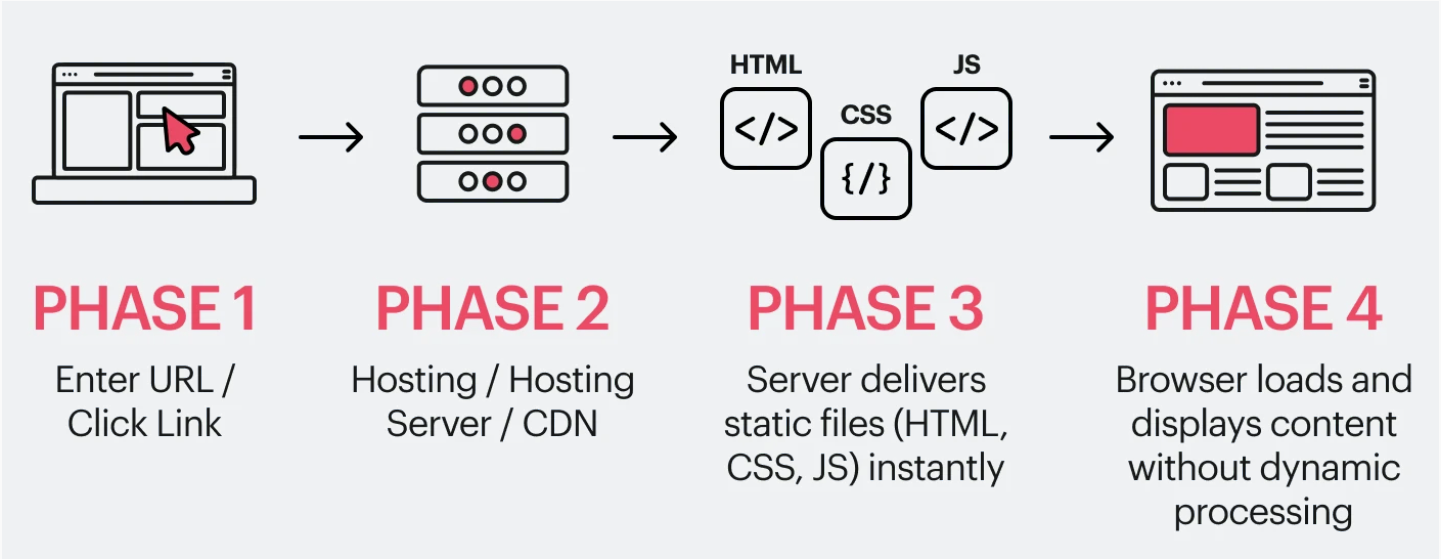

Figure 1: Static Webpage

HyperText Markup Language i

Let's break that down:

- **HyperText:** This refers to text that contains links ("hyperlinks") to other texts (or pages). This is the fundamental concept that connects the entire web.
- **Markup Language:** This is *not* a programming language.
  - A **programming language** *does* things (e.g., `if (x > 5) { do_something() }`).
  - A **markup language** *describes* things. It uses "tags" to define the structure and meaning of the content.

HyperText Markup Language ii

Example:

```
<!-- This is NOT programming. It's a description. -->
<!-- It says: "This text is a level-one heading." -->
<h1>This is my main title</h1>

<!-- It says: "This text is a paragraph." -->
<p>This is a block of text.</p>
```

The Evolution to HTML5 i

Older HTML (HTML4):

We only had one generic container: `<div>`. This led to code that was hard to read for both developers and machines (like screen readers or search engines).

```
<!-- HTML4: What does this content MEAN? -->
<div id="header">
  <div id="nav">...</div>
</div>
<div id="main-content">
  <div class="article">...</div>
</div>
<div id="footer">...</div>
```

The Evolution to HTML5 ii

Modern HTML (HTML5):

HTML5 introduced **semantic tags** that describe their *meaning* and *purpose*.

```
<!-- HTML5: The meaning is clear! -->
<header>
  <nav> ... </nav>
</header>
<main>
  <article> ... </article>
</main>
<footer> ... </footer>
```

HTML4 vs HTML5

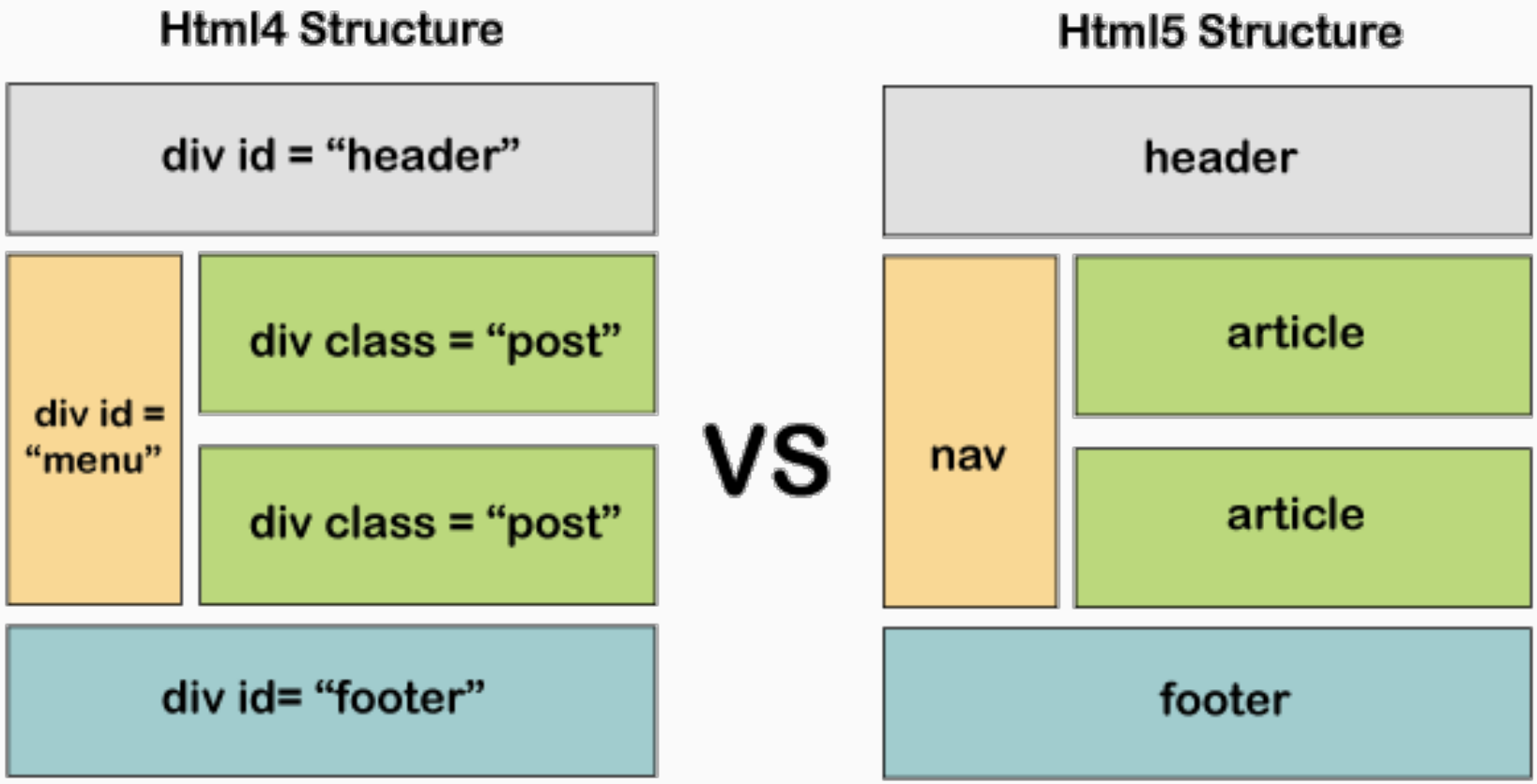

Figure 2: HTML4 vs HTML5

Why HTML5 Semantics Matter i

Using tags like `<header>`, `<main>`, and `<article>` is critical for:

1. **Accessibility:**
    - Screen readers (used by visually impaired users) can understand the page structure.
    - They can say “Now entering the main content” or “Skipping to navigation.” This is impossible with generic `<div>` tags.
2. **Search Engine Optimization (SEO):**
    - Search engines (like Google) can better understand what your content is about.

Why HTML5 Semantics Matter ii

- Content inside an `<article>` tag is clearly an article. Content in `<nav>` is clearly for navigation. This helps your page rank correctly.

3. **Developer Readability:**
    - Your code becomes cleaner, more organized, and easier to maintain.

The Base HTML5 Skeleton i

Every modern HTML file should have this basic structure:

```
<!DOCTYPE html>
<!-- This DOCTYPE tells the browser it's a modern HTML5 document -->

<html lang="en">
<!-- The <html> tag wraps all content. 'lang="en"' helps accessibility. -->

  <head>
    <!-- The <head> contains "meta" data (data ABOUT the page) -->
    <!-- It is NOT visible to the user. -->
    <meta charset="UTF-8"> <!-- Tells the browser to use UTF-8 text encoding -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My First Webpage</title> <!-- Shows in the browser tab -->
    <link rel="stylesheet" href="style.css"> <!-- Links to our CSS -->
  </head>

  <body>
    <!-- The <body> contains the *visible* content of the page -->
    <!-- All of our structure (header, main, etc.) goes here. -->
  </body>

</html>
```

The Base HTML5 Skeleton ii

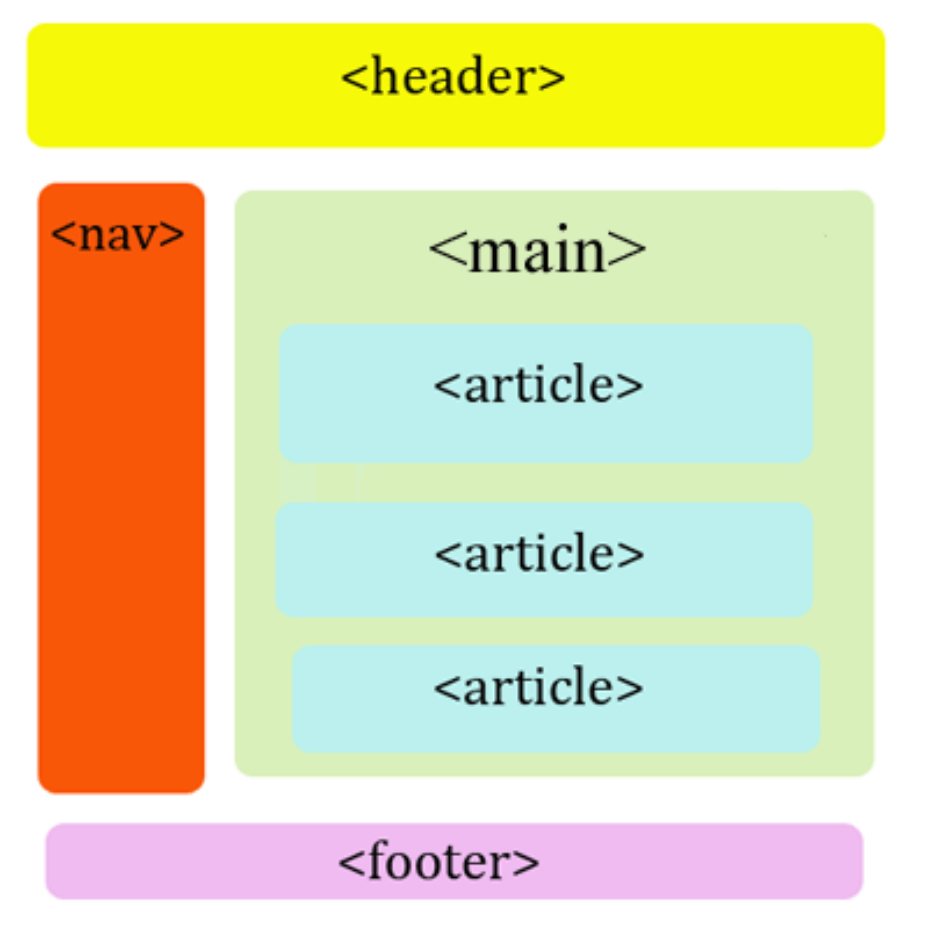

Figure 3: HTML5

Header: <header> i

- **Purpose:** Contains introductory content for the entire page (or an <article>).
- **Not** to be confused with the <head> tag!
- **Typical Content:** The site logo, the main site title, search bar, and often the main navigation.

```
<header>
  <img src="/logo.png" alt="My Site Logo">
  <h1>My Awesome Website</h1>
  <!-- The <nav> often lives inside the <header> -->
  <nav>
    ...
  </nav>
</header>
```

Navigation: `<nav>` i

- **Purpose:** Specifically for “major navigational blocks.” Use it for your main site navigation, table of contents, etc.
- **Don’t** use it for *every* list of links (e.g., links in a `<footer>`).
- It’s common to put an unordered list (`<ul>`) inside it.

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

The Core Content: `<main>` i

- **Purpose:** This tag is for the **main, unique content** of the page.
- It should contain the “point” of the page (e.g., the blog post, the product details, the “About Us” text).
- **Rule:** There should be **only one** `<main>` tag per page.
- It should *not* contain repeating content like sidebars, headers, or footers.

The Core Content: <main> ii

```
<body>
  <header> ... </header>
  <nav> ... </nav>

  <!-- This is the unique content for THIS page -->
  <main>
    <h2>About Our Company</h2>
    <p>We were founded in 2025 ... </p>
  </main>

  <footer> ... </footer>
</body>
```

Organizing Content: `<article>` vs. `<section>` i

This is the most confusing part of HTML5, but it's simple:

`<article>`

- **Purpose:** For a **complete, self-contained** piece of content that could (in theory) be distributed on its own (like an RSS feed).
- **Test:** Could you print *just* this part, and would it make sense?
- **Examples:** A blog post, a news story, a forum comment, a product card.

Organizing Content: `<article>` vs. `<section>` ii

`<section>`

- **Purpose:** For a **thematic grouping** of content *within* a page (or *within* an article).
- **Test:** Is this a “chapter” or “subsection” of a larger whole?
- **Examples:** A “Contact Us” area, a “Top Stories” block, Chapter 1 of an article.

Sidenotes: `<aside>` i

- **Purpose:** For content that is “tangentially related” to the content around it.
- If you removed it, the main content would still make perfect sense.
- **Typical Use:** A sidebar.
- **Examples:** Related links, a “pull quote” from an article, author biography, advertisements.

Sidenotes: <aside> ii

```
<main>
  <article>
    <h1>My Blog Post</h1>
    <p> ... the main text ... </p>
  </article>

  <aside>
    <h3>Related Posts</h3>
    <ul>
      <li><a href=" ... ">Another Post</a></li>
    </ul>
  </aside>
</main>
```

The End: <footer> i

- **Purpose:** Represents the “footer” for its nearest “sectioning root” (which is usually the <body> itself).
- **Typical Content:** Copyright information, secondary navigation (privacy policy, sitemap), contact links.

```
<footer>
  <p>&copy; 2025 My Awesome Website</p>
  <ul>
    <li><a href="/privacy">Privacy Policy</a></li>
    <li><a href="/sitemap">Sitemap</a></li>
  </ul>
</footer>
```

HTML5 Media: Images `<img>` i

- The `<img>` tag is used to embed images. This includes `.jpg`, `.png`, `.svg`, and animated `.gif` files.
- It is an "empty" tag, meaning it has no closing tag.
- `src` **(source):** The path or URL to the image file. This is **required**.
- `alt` **(alternative text):** A description of the image. This is **critical** for accessibility (for screen readers) and for SEO. It will also display if the image fails to load.

HTML5 Media: Images <img> ii

```
<!-- A standard image -->
<img src="images/my-photo.jpg" alt="A photo of me on a mountain.">

<!-- An animated GIF uses the same tag -->
<img src="gifs/funny-reaction.gif" alt="A cat looking surprised.">

<!-- An image loaded from an external website -->
<img src="[https://example.com/logo.png](https://example.com/logo.png)" alt="Example Company
```

HTML5 Media: Audio `<audio>` i

- Before HTML5, audio required plugins like Flash. Now it's native!
- The `<audio>` tag embeds sound content.
- `controls`**:** This boolean attribute adds the browser's default play/pause/volume controls. **You should always include this.**
- `<source>` **Tag:** You can provide multiple file formats. The browser will play the *first* one it supports. This is crucial for cross-browser compatibility.

HTML5 Media: Audio <audio> ii

```html
<!-- Simple audio file with controls -->
<audio controls src="music/my-song.mp3">
  Your browser does not support the audio element.
</audio>

<!-- Best practice: Using the <source> tag -->
<audio controls>
  <source src="music/my-song.ogg" type="audio/ogg">
  <source src="music/my-song.mp3" type="audio/mpeg">
  Your browser does not support the audio element.
</audio>
```

HTML5 Media: Video `<video>` i

- Like `<audio>`, the `<video>` tag replaced Flash for video content.
- `controls`: Again, this adds the default play/pause/fullscreen controls.
- `width` / `height`: You can set the dimensions of the video player.
- `<source>` **Tag:** Just like audio, this is the best practice for providing multiple video formats (like `.mp4` and `.webm`) to support all browsers.

HTML5 Media: Video <video> ii

```
<!-- Simple video file with controls -->
<video controls width="640" height="480" src="movies/my-video.mp4">
  Your browser does not support the video tag.
</video>

<!-- Best practice: Using the <source> tag -->
<video controls width="640" height="480">
  <source src="movies/my-video.webm" type="video/webm">
  <source src="movies/my-video.mp4" type="video/mp4">
  Your browser does not support the video tag.
</video>
```

Putting It All Together i

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>My Semantic Page</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

  <header>
    <h1>My Site</h1>
    <nav>
      <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/blog">Blog</a></li>
      </ul>
    </nav>
  </header>

  <main>
    <article>
      <h2>My First Blog Post</h2>
      <p>This is the main text of my post.</p>
      <figure>
```

Putting It All Together ii

```
        <img src="images/post-image.jpg" alt="A key image for my post.">
        <figcaption>This is an image caption.</figcaption>
      </figure>

      <section>
        <h3>Part 1</h3>
        <p>The first part of the post.</p>
      </section>
    </article>
  </main>

  <aside>
    <h3>About the Author</h3>
    <p>I love writing blog posts.</p>
  </aside>

  <footer>
    <p>&copy; 2025 My Site</p>
  </footer>

</body>
</html>
```

What is CSS?

- **C**ascading **S**tyle **S**heets.
- CSS provides the **style** (the “skin”) for your HTML (the “skeleton”).
- It controls colors, fonts, layout, spacing, and animations.
- HTML and CSS are **separate** languages. This is a core principle called **Separation of Concerns**.
    - HTML is for *content and structure*.
    - CSS is for *presentation*.

The “C”: Cascading

- “Cascading” refers to the set of rules that determines which style is applied if multiple rules conflict.
- **The Cascade (simplified):**
  1. **Origin:** Styles from a developer’s stylesheet (your `style.css`) override the browser’s default styles.
  2. **Specificity:** A more *specific* selector wins. `#my-id` is more specific than `.my-class`, which is more specific than p.
  3. **Source Order:** If two selectors have the *same* specificity, the one that appears **last** in the file wins.

```
/* style.css */
p { color: blue; }
p { color: red; } /* This one wins (Source Order) */

.my-p { color: green; }
p { color: purple; } /* .my-p is more specific, so green wins! */
```

The “S”: Selectors & Properties i

- CSS works on a simple “Selector + Declaration” model.

```
/* Selector     Declaration Block  */
/* |           |-⊣ */
    h1 {
/* Property   Value    */
/* |          |      */
      color: blue;
      font-size: 24px;
    }
/* |⊣ */
/* This whole thing is a "Ruleset"           */
```

- **Selectors (The “Who”):**
  - **Element:** `p` (selects all `<p>` tags)
  - **Class:** `.my-class` (selects all tags with `class="my-class"`)
  - **ID:** `#my-id` (selects the *one* tag with `id="my-id"`)

Methods of Integration i

External (Best Practice)

- You write all your CSS in a separate `style.css` file.
- You link it in the HTML `<head>`.
- **Pros:**
  - **Reusable:** One CSS file can style 1000 HTML pages.
  - **Maintainable:** To change the site color, you edit *one* file.
  - **Caching:** The browser downloads `style.css` once and saves (caches) it, making other pages load faster.

```
<!-- in index.html <head> -->
<link rel="stylesheet" href="styles.css">
```

Methods of Integration ii

Internal / Embedded

- You write your CSS inside a `<style>` tag directly in the HTML `<head>`.
- **Pros:**
  - Useful for single-page demos or quick tests.
- **Cons:**
  - Only affects this *one* HTML file.
  - Cannot be cached separately.
  - Makes the HTML file large and messy.

```
<!-- in index.html <head> -->
<style>
  body {
    background-color: #f0f0f0;
  }
  h1 {
    color: teal;
  }
</style>
```

Inline (Avoid!)

- You write CSS directly inside an HTML tag's `style` attribute.
- This mixes content and style, violating the "Separation of Concerns."
- **Cons:**
  - Extremely hard to maintain.
  - Highest possible specificity (it "wins" against all other styles), which causes hard-to-debug problems.
  - Clutters your HTML.

```
<!-- in index.html <body> -->
<!-- This is bad practice! -->
<p style="color: red; font-size: 12px;">
  This is a paragraph.
</p>
```

Static (HTML/CSS) - “The Brochure”

- **Server’s Job:** Find the requested file (e.g., `about.html`) and send it.
- **Client’s (Browser’s) Job:** Receive the file and render it.
- The content is **fixed**.

Dynamic (JS) - “The Application”

- **Server’s Job:** Send a *shell* HTML file and a bundle of JavaScript (`app.js`).
- **Client’s (Browser’s) Job:**
  1. Receives the shell HTML.
  2. Receives and *executes* the `app.js` file.
  3. The JavaScript **manipulates the DOM** (Document Object Model) to build the page, show/hide elements, and create interactivity.
  4. This is how React, Angular, and Vue work.

Client-Side Dynamics: `fetch()`

- JavaScript makes a “dynamic” page possible by fetching data *after* the page has loaded.
- It uses the `fetch()` API (this used to be called AJAX).
- **Model:** This is a **Request-Response** (or “Pull”) model.

Flow:

1. User clicks a “Load Posts” button.
2. JavaScript runs: `fetch('/api/posts')`
3. The browser sends a *new* HTTP request to the server.
4. The server sends back *only* data (usually in **JSON** format).
5. The JavaScript receives the JSON, loops over it, and creates new HTML to display the posts.

- **Key Idea:** The page content changes *without a full page reload*.

The Limit of `fetch()`: Polling

- `fetch()` is great, but it's "client-initiated." The client must *ask* for data.
- **Problem:** How do you build a real-time chat application?
- **Bad Solution (Polling):** The client has to ask the server every 2 seconds, "Any new messages? ... Any new messages? ... Any new messages?"
- This is *highly* inefficient and floods your server with requests.

Real-Time Dynamics: WebSockets i

- **The Solution:** WebSockets.
- A WebSocket is *not* a "request." It's a **persistent, two-way connection** (like a phone call).
- **Flow:**
  1. The client sends a special "Upgrade" request to the server.
  2. The server accepts, and the connection is "upgraded" from HTTP to a WebSocket (`ws://` or `wss://`).
  3. This connection *stays open*.
- Now, the **server can *push* data** to the client at any time, instantly.
- **Chat App Example:**
  1. User A sends a message.

Real-Time Dynamics: WebSockets ii

2. The server receives it (via WebSocket).
3. The server *pushes* that message to User B (via their WebSocket).
4. The message appears on User B's screen instantly. No polling needed.

Real-Time Dynamics: MQTT

- You may also hear about **MQTT** (Message Queuing Telemetry Transport).
- This is *not* a browser-native technology like WebSockets.
- It is an extremely lightweight **publish/subscribe (pub/sub)** protocol, popular for **IoT** (Internet of Things) devices.
- **Pub/Sub Model:**
  - A device "publishes" a message to a "topic" (e.g., `home/living_room/temperature`).
  - Any other device "subscribed" to that topic receives the message instantly.
- **How it relates to web:** You can use a server (a "broker") that *bridges* MQTT to WebSockets, allowing a web dashboard (in your browser) to subscribe to topics and get real-time updates from IoT devices.

How a Browser *Really* Accesses a Page i

A detailed, 13-step process:

1. **You type** `http://example.com` and hit Enter.
2. **Browser:** "What is the IP address for `example.com`?" It checks its local cache.
3. **OS:** Browser didn't know. The Operating System checks *its* cache (e.g., `hosts` file).
4. **DNS Resolver:** OS didn't know. The request goes to a DNS Resolver (e.g., your router, or 8.8.8.8). The resolver finds the IP address (e.g., `93.184.216.34`) for `example.com`.

How a Browser *Really* Accesses a Page ii

5. **TCP Handshake:** The browser now knows the IP. It opens a TCP connection (a "socket") to `93.184.216.34` on **port 80** (for HTTP) or **port 443** (for HTTPS). This involves a "three-way handshake" (SYN, SYN-ACK, ACK).
6. **HTTP Request:** Once the connection is open, the browser sends a plain-text HTTP request:

```
GET / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; ...)
Accept: text/html, ...
```

7. **The Request Hits Your Server:** The request arrives at the server's IP (`93.184.216.34`).
8. **The Docker Part:**

How a Browser *Really* Accesses a Page iii

- Your `docker-compose.yml` mapped port 80 on the host machine to port 80 in the **Nginx container**.
- Docker's networking layer instantly forwards this request packet into the container.

9. **The Nginx Part:**
    - Nginx (running in the container) receives the `GET /` request.
    - It checks its configuration (e.g., `nginx.conf`).
    - It sees it should serve files from its `root` directory, which is `/usr/share/nginx/html`.

How a Browser *Really* Accesses a Page iv

- Thanks to your `docker-compose.yml` **volume mount** (`./my-html-folder:/usr/share/nginx/html`), when Nginx looks in its *container* folder, it's *actually* reading from your *host machine's* folder.
- Nginx finds the file `index.html`.

10. **HTTP Response:** Nginx sends a plain-text HTTP response back over the same TCP connection:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234

<!DOCTYPE html>... (your entire HTML file)
```

11. **Browser Parsing:** The browser receives this HTML. It starts reading it from top to bottom.

How a Browser *Really* Accesses a Page v

12. **Subsequent Requests:** The browser reads the `<head>` and finds: `<link rel="stylesheet" href="style.css">`
    - **It pauses parsing** and goes back to **Step 5** to make a *brand new HTTP request* to fetch `style.css`.
    - It does the same for `<img src=" ... ">`, `<script src=" ... ">`, etc.
13. **Render:** Once all files are fetched, the browser "paints" the page on your screen.

Nginx + Docker Compose Example i

This `docker-compose.yml` file is the recipe for your server in Step 8-9.

```yaml
# docker-compose.yml
version: '3.8'

services:
  # 'webserver' is just a name we choose
  webserver:
    # Use the official, lightweight Nginx image
    image: nginx:alpine

    # Map [Host Port]:[Container Port]
    # "Any traffic on my machine's port 8080 ...
    #  ... should be sent to this container's port 80"
    ports:
      - "8080:80"

    # Mount [Host Folder]:[Container Folder]
    # "Make my local './my-site' folder ...
    #  ... appear inside the container at '/usr/share/nginx/html'"
    volumes:
```

Nginx + Docker Compose Example ii

```
    - ./my-site:/usr/share/nginx/html:ro

  # ':ro' means 'read-only', a good security practice.
```

- **To Run:** `docker compose up`
- **To Access:** `http://localhost:8080`

A (Forward) Proxy

First, let's define a "forward proxy," which you might use at school or work.

- **Hides the Client.**
- You (client) -> Proxy -> Internet
- The Internet (e.g., Google) sees the request coming from the *Proxy's IP*, not yours.
- Its purpose is to filter/monitor *your* traffic.

A Reverse Proxy

A **reverse proxy** is the opposite.

- **Hides the Server(s).**
- Internet (client) -> Reverse Proxy -> Your App Servers
- You (the client) only know the IP of the reverse proxy. You have *no idea* what servers are running behind it.
- This is the standard for *all* modern web applications.

Why Use a Reverse Proxy? i

A reverse proxy (like Nginx) is a “traffic cop” that provides:

1. **Request Routing:**
    - It can look at the URL and send the request to different *internal* servers.
    - `example.com/` -> goes to your `frontend-app` container.
    - `example.com/api/` -> goes to your `backend-api` container.
2. **Load Balancing:**
    - What if your backend is too busy? You can run 10 copies of it.

Why Use a Reverse Proxy? ii

- The reverse proxy will *distribute* the traffic between all 10 copies (“load balancing”) so no single server is overwhelmed.

3. **SSL Termination:**
   - Handling HTTPS (encryption) is computationally expensive.
   - Let the reverse proxy do all the hard work of encrypting/decrypting (this is “SSL termination”).
   - Behind the proxy, your internal apps can communicate using simple, fast HTTP.
4. **Caching:**

Why Use a Reverse Proxy? iii

- The proxy can store copies of static files (images, CSS) and serve them directly, reducing load on your app servers.

Reverse Proxy: Docker Compose Example

This `docker-compose.yml` has 3 services. **Only the proxy is public.**

```yaml
version: '3.8'
services:
  # 1. The Reverse Proxy (Public)
  proxy:
    image: nginx:alpine
    ports:
      - "80:80" # This is the ONLY public port
    volumes:
      # We give it a custom config file
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
  # 2. The Frontend App (Private)
  frontend:
    image: my-frontend-app # Your React/Vue/etc app
    # NO ports. It's hidden.
  # 3. The Backend API (Private)
  backend:
    image: my-backend-api # Your Python/Node/Java API
    # NO ports. It's hidden.
```

Reverse Proxy: Nginx Config Example

This is the `nginx.conf` file that tells the proxy how to route traffic.

```
# nginx.conf
events {}
http {
  server {
    listen 80; # Listen on the public port 80

    # Rule 1: If URL is /api/...
    location /api/ {
      # Pass the request to the 'backend' service
      # (Docker's internal DNS resolves 'backend' to the container's IP)
      proxy_pass http://backend:5000; # Assuming API runs on port 5000
    }

    # Rule 2: For everything else (/)
    location / {
      # Pass the request to the 'frontend' service
      proxy_pass http://frontend:3000; # Assuming frontend runs on 3000
    }
  }
}
```

What is CORS?

The Problem: Same-Origin Policy (SOP)

- This is a **critical security feature** built into *all* modern browsers.
- It states: A script on a webpage can only make requests to **the *same origin* it was loaded from.**
- **What is an "Origin"?**
  - `Protocol` + `Domain` + `Port`
  - `http://example.com:80` and `https://example.com:443` are **DIFFERENT** origins (Protocol).
  - `http://site.com` and `http://api.site.com` are **DIFFERENT** origins (Domain).
  - `http://localhost:3000` and `http://localhost:5000` are **DIFFERENT** origins (Port).

Why does SOP exist?

To stop `http://evil.com` (which you visited) from using JavaScript to make a `fetch()` request to `http://my-bank.com` (which you are logged into) and steal your money.

The Solution: CORS

- **C**ross-**O**rigin **R**esource **S**haring
- The SOP is good, but it *also* blocks our *legitimate* `http://localhost:3000` (frontend) from talking to `http://localhost:5000` (backend).
- **CORS** is the mechanism to *relax* the SOP.
- It's **not** a setting in the browser. It's a set of **HTTP headers** that the **SERVER** sends back to the browser.
- By sending these headers, the server tells the browser: "It's okay, I give permission to *that specific origin* to access my resources."

Example:

1. Frontend (`localhost:3000`) makes `fetch('http://localhost:5000/api/users')`.
2. Browser sees it's a "cross-origin" request and pauses it.
3. Browser sends a "preflight" `OPTIONS` request to `http://localhost:5000`.
4. Your **backend server** must respond with the "permission" headers:

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: http://localhost:3000
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Allow-Headers: Content-Type
```

5. Browser sees this, says "OK, permission granted," and sends the *real* `GET` request.

Further Resources

- **W3Schools:**
  - A great site for hands-on tutorials and examples.
  - https://www.w3schools.com/
- **MDN (Mozilla Developer Network):**
  - The single best resource for learning and looking up HTML, CSS, and JS.
  - developer.mozilla.org
- **Nginx & Docker Docs:**
  - The official documentation is your best friend.
  - docs.docker.com
  - nginx.org/en/docs