Programação Avançada para a Internet

Mestrado em Engenharia de Software

Web Programming Review

Introduction: The Three Pillars of the Web

This presentation will provide a detailed review of the three core technologies that power every website.

- **HTML:** The **structure** and skeleton of the page.
- **CSS:** The **presentation** and styling that brings it to life.
- **JavaScript:** The **interactivity** and logic that makes it dynamic.

Part 1: HTML

HTML Document Structure Explained

Every HTML page is built on this essential boilerplate.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
     <title>Page Title</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>My Awesome Webpage</h1>
    <p>Content goes here...</p>
</body>
</html>
```

The Power of Semantic HTML

Using elements based on their **meaning**, not their appearance, is critical for **accessibility** and **SEO**. A screen reader or search engine can understand the structure of a semantic page far better.

Non-Semantic Example (Avoid This):

```
<div id="header">...</div>
<div class="main-content">
  <div class="article">...</div>
</div>
<div id="footer">...</div>
```

The Power of Semantic HTML

Semantic Example (Best Practice):

```
<header>...</header>
<main>
  <article>...</article>
</main>
<footer>...</footer>
```

Common semantic tags include `<header>`, `<nav>`, `<main>`, `<section>`, `<article>`, `<aside>`, and `<footer>`.

Essential Content Elements

These are the building blocks for most of your page content.

- **Headings:** `<h1>` to `<h6>` define a content hierarchy. Use `<h1>` only once per page for the main title.
- **Text:** `<p>` for paragraphs, `<strong>` for importance, `<em>` for emphasis, `<blockquote>` for quoted sections.
- **Links:** `<a>` creates hyperlinks. The `href` attribute is the destination.

```
<a href="/about.html">Internal Link</a>
<a href="#section2">Page Anchor</a>
```

Essential Content Elements

- **Images:** `<img>` requires `src` (source) and `alt` (alternative text for accessibility).

```
<figure>
  <img src="logo.png" alt="My company logo." width="200" height="100">
  <figcaption>Our official company logo.</figcaption>
</figure>
```

Structuring Data with Tables

Tables are for **tabular data**, not for page layout. Use `<thead>` and `<tbody>` to semantically structure your table.

```
<table>
  <thead>
    <tr><th>Product</th><th>Price</th><th>Availability</th></tr>
  </thead>
  <tbody>
    <tr><td>Widget A</td><td>$19.99</td><td>In Stock</td></tr>
    <tr><td>Widget B</td><td>$29.99</td><td>Out of Stock</td></tr>
  </tbody>
</table>
```

Advanced Forms for User Input

Forms are essential for user interaction. You should be familiar with a variety of input types and structural elements.

```
<form action="/signup" method="POST">
  <label for="email">Email:</label>
  <input type="email" id="email" name="user_email" required>
  <label for="subscription">Plan:</label>
  <select id="subscription" name="user_plan">
    <option value="free">Free Tier</option>
    <option value="premium" selected>Premium Tier</option>
  </select>
  <fieldset>
    <legend>Interests</legend>
    <input type="checkbox" id="dev" name="interests" value="development">
    <label for="dev">Development</label><br>
    <input type="checkbox" id="design" name="interests" value="design">
    <label for="design">Design</label>
  </fieldset>
  <button type="submit">Sign Up</button>
</form>
```

Part 2: CSS - The Language of Style

Selectors, Specificity, and the Cascade

These three concepts determine which CSS rules apply to an element.

- **Selectors:** Target elements. `ID > Class > Element`. (`#my-id` is more specific than `.my-class`).
- **Specificity:** A score/weight that determines which rule wins in a conflict.
- **Cascade:** The order of rules matters. If specificity is equal, the last defined rule wins. Inline styles override stylesheets, and `!important` overrides everything (use with extreme caution).

Selectors, Specificity, and the Cascade

Specificity Example:

```
/* Specificity: 0,0,1 (Element) */
p { color: black; }

/* Specificity: 0,1,0 (Class) - Wins over the p selector */
.highlight { color: orange; }

/* Specificity: 1,0,0 (ID) - Wins over both selectors above */
#main-paragraph { color: blue; }
```

The CSS Box Model

Every element is a box. `box-sizing: border-box` is a modern standard that makes layout calculations more intuitive.

```
.my-element {
  /* This setting is highly recommended for all elements */
  box-sizing: border-box;

  width: 200px; /* The total width, including padding and border */
  padding: 20px; /* Space inside the border */
  border: 5px solid black; /* The line around the padding */
  margin: 15px; /* Space outside the border */
}
```

Without `border-box`, the actual width would be `200px + 20px + 20px + 5px + 5px = 250px`.

Modern Layout: CSS Flexbox

Flexbox is a **one-dimensional** model perfect for aligning items in a row or column. It's ideal for component-level layouts like navigation bars, card components, and form controls.

```
.nav-container {
  display: flex; /* Establishes a flex context */
  flex-direction: row; /* Main axis is horizontal */
  justify-content: space-between; /* Distributes items along the main axis */
  align-items: center; /* Aligns items along the cross (vertical) axis */
}
```

Use Flexbox when: You need to align items in a single dimension (horizontally or vertically).

Modern Layout: CSS Grid

Grid is a **two-dimensional** model for creating complex row and column-based layouts. It is ideal for the overall page structure.

```
.page-wrapper {
  display: grid;
  /* Creates a 200px sidebar and a main content area that takes the remaining space */
  grid-template-columns: 200px 1fr;
  grid-template-rows: auto 1fr auto;
  grid-template-areas:
    "header header"
    "sidebar main"
    "footer footer";
  gap: 20px; /* Defines the space between grid items */
}
.header { grid-area: header; }
.sidebar { grid-area: sidebar; }
/* etc. */
```

Use Grid when: You need to control both rows and columns for a page-level layout.

Responsive Design with Media Queries

Media queries apply CSS rules based on device characteristics like screen width, enabling responsive design. A **mobile-first** approach is the industry standard.

```
/* Base styles (for small screens) */
.container {
  width: 100%;
}
/* Styles for tablets and larger */
@media (min-width: 768px) {
  .container {
    display: flex;
    justify-content: space-between;
  }
}
/* Styles for desktops and larger */
@media (min-width: 1200px) {
  .container {
    max-width: 1140px;
    margin: 0 auto;
  }
}
```

CSS Frameworks: Bootstrap

Bootstrap is a **component-based** framework. It provides pre-built, pre-styled components (like buttons, cards, modals) that you can use to quickly assemble a user interface.

Purpose: Rapid development and prototyping with a consistent look and feel.

```
<!-- Bootstrap Button -->
<button type="button" class="btn btn-primary">Primary Button</button>

<!-- Bootstrap Card -->
<div class="card" style="width: 18rem;">
  <div class="card-body">
    <h5 class="card-title">Card Title</h5>
    <p class="card-text">Some quick example text.</p>
    <a href="#" class="btn btn-primary">Go somewhere</a>
  </div>
</div>
```

CSS Frameworks: Tailwind CSS

Tailwind is a **utility-first** framework. It provides low-level utility classes that you compose directly in your HTML to build custom designs. You are not given pre-built components, but rather the tools to build your own.

Purpose: Building highly customized user interfaces without writing custom CSS.

```
<!-- Tailwind Button -->
<button class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
  Primary Button
</button>
<!-- Tailwind Card -->
<div class="max-w-sm rounded overflow-hidden shadow-lg bg-white">
  <div class="px-6 py-4">
    <div class="font-bold text-xl mb-2">Card Title</div>
    <p class="text-gray-700 text-base">Some quick example text.</p>
  </div>
  <div class="px-6 py-4">
    <a href="#" class="bg-blue-500 text-white font-bold py-2 px-4 rounded">
      Go somewhere
    </a>
  </div>
</div>
```

Bootstrap vs. Tailwind CSS

The choice depends on the project's goals.

Feature	Bootstrap	Tailwind CSS
Philosophy	Component-Based	Utility-First
Customization	Less flexible; requires overriding styles	Highly flexible; built for customization
Development Speed	Very fast for standard UIs	Slower initially, but faster for custom designs
Learning Curve	Easy	Steeper (requires learning the utility classes)
Result	Can look "generic" if not customized	Unique, custom designs

Component Options for Tailwind CSS

Since Tailwind provides utilities, not UI components, several libraries have emerged to fill this gap for faster development:

- **Styled Component Libraries**: These offer pre-designed components built with Tailwind classes, similar to Bootstrap but using the Tailwind ecosystem.
    - **Examples**: DaisyUI, Flowbite, Preline UI.
- **Headless UI Libraries**: These provide completely unstyled, accessible components with all the necessary functionality (JavaScript). You apply Tailwind classes to style them from scratch.
    - **Examples**: Headless UI, Radix UI.

Part 3: JavaScript - The Engine of Interaction

Variables, Scope, and Data Types

- **Declaration:** `let` (re-assignable, block-scoped), `const` (not re-assignable, block-scoped), `var` (function-scoped, generally avoid).
- **Scope:** Determines the accessibility of variables. Modern JS uses block scope (`{...}`).
- **Data Types:**
  - **Primitives:** `String`, `Number`, `Boolean`, `null`, `undefined`, `Symbol`
  - **Structural:** `Object` (key-value pairs), `Array` (ordered list)

Variables, Scope, and Data Types

```
const name = "Alice"; // String
let age = 30;         // Number
const isStudent = true; // Boolean
const user = {        // Object
  id: 1,
  username: "alice_dev"
};
const skills = ["HTML", "CSS", "JavaScript"]; // Array
```

Functions: Reusable Blocks of Code

Functions are fundamental. You should be comfortable with declarations, expressions, and arrow functions.

```
// Function Declaration
function greet(name) {
  return `Hello, ${name}!`;
}
// Function Expression
const sayGoodbye = function(name) {
  return `Goodbye, ${name}.`;
};
// Arrow Function (concise syntax)
const add = (a, b) => {
  return a + b;
};
// Arrow Function (implicit return)
const subtract = (a, b) => a - b;
```

DOM Manipulation: Bringing Pages to Life

The Document Object Model (DOM) is the browser's representation of your HTML. JavaScript can manipulate it to create dynamic experiences.

Key Methods:

- **Selecting:** `document.querySelector()`, `document.getElementById()`
- **Modifying:** `.textContent`, `.innerHTML`, `.style`
- **Changing Attributes:** `.setAttribute()`, `.getAttribute()`
- **Managing Classes:** `.classList.add()`, `.classList.remove()`, `.classList.toggle()`
- **Creating/Adding:** `document.createElement()`, `parentElement.appendChild()`

DOM Manipulation Example

```
<div id="container"></div>
<button id="actionBtn">Add Item</button>
```

```
const container = document.querySelector('#container');
const button = document.querySelector('#actionBtn');
let itemCount = 0;

button.addEventListener('click', () => {
  itemCount++;
  // Create a new <p> element
  const newItem = document.createElement('p');

  // Set its content and class
  newItem.textContent = `This is item number ${itemCount}.`;
  newItem.classList.add('item');

  // Append it to the container
  container.appendChild(newItem);
});
```

Events: Responding to User Actions

Event listeners wait for a specific user action (like a 'click' or 'submit') on an element and execute a function in response.

```
const form = document.querySelector('#contact-form');

form.addEventListener('submit', (event) => {
  // Prevent the default form submission (which reloads the page)
  event.preventDefault();

  console.log('Form submitted!');
  // Here you would add logic to handle the form data
});
```

Common events include `click`, `mouseover`, `keydown`, `keyup`, and `submit`.

Asynchronous JavaScript: `fetch` and Promises

Modern web applications frequently need to load data from a server without freezing the user interface. `fetch` is the standard API for this. It returns a **Promise**—an object representing the eventual completion (or failure) of an async operation.

The `async/await` syntax makes working with Promises much cleaner.

```
const API_URL = 'https://jsonplaceholder.typicode.com/todos/';
async function fetchData() {
  try {
    const response = await fetch(API_URL); // Pause until the request completes
    if (!response.ok) { throw new Error(`HTTP error! Status: ${response.status}`); }
    const data = await response.json(); // Pause until the JSON is parsed
    console.log(data); // Now you can use the data
  } catch (error) {
    console.error('Could not fetch data:', error);
  }
}
fetchData();
```

Part 4: The Development Ecosystem

Browser Developer Tools

Your most important debugging tool. You should be proficient with:

- **Elements Panel:** Inspecting and live-editing HTML and CSS.
- **Console:** Viewing JavaScript logs, errors, and running code snippets.
- **Network Tab:** Monitoring all network requests, their status, and payload.
- **Debugger/Sources:** Setting breakpoints in your JavaScript to step through code execution.

Mastering the dev tools is non-negotiable for an advanced developer.

The JavaScript Ecosystem: Node.js & NPM

- **Node.js:** A runtime to execute JavaScript **outside of the browser**.
- **NPM (Node Package Manager):** A command-line tool and package registry (npmjs.com) for managing third-party code.
- **Crucially**, even for front-end projects where you don't write a Node.js server, this ecosystem is indispensable. Tools like Vite, React, linters, and formatters are all installed and managed as NPM packages.
- While `npm` is the default, other managers like **`yarn`** and **`pnpm`** offer improvements and use the same NPM registry.

The JavaScript Ecosystem: `package.json`

The `package.json` file is the heart of a Node project. It defines project metadata and lists its dependencies.

```
{
  "name": "my-project",
  "version": "1.0.0",
  "scripts": {
    "dev": "vite",
    "build": "vite build"
  },
  "dependencies": {
    "react": "^18.2.0"
  }
}
```

- `npm install`: Installs all dependencies listed in `package.json`.
- `npm run dev`: Runs the script associated with the "dev" key.

Conclusion: Ready for the Next Level

A strong command of these fundamental concepts is the prerequisite for tackling advanced web development. As you move forward, you'll build upon this foundation to learn about:

- **JavaScript Frameworks:** (React, Angular, Vue)
- **Server-Side Development & APIs:** (Node.js, Express)
- **Build Tools & Bundlers:** (Vite, Webpack)
- **Advanced CSS Techniques:** (Animations, Custom Properties)
- **Testing and Deployment**

Keep practicing, stay curious, and good luck!