

# Modern Frontend Web Development

---

HTML5, CSS3, JavaScript, Tools, and Web APIs for new web developers



Course Repository:

<https://github.com/chrisminnick/modern-frontend-web-dev>

**Version:** 3.0.0

**Date:** September 2025

**Author:** Chris Minnick

**Copyright © 2025 WatzThis?**

All rights reserved.

**Website:** <https://www.watzthis.com>



# Module 1: Introduction to Web Development Fundamentals

# Course Overview

---

## Modern Frontend Web Development

### What You'll Learn:

- HTML5 semantic markup and modern standards
- CSS3 with Grid, Flexbox, and responsive design
- JavaScript ES6+ with modern programming patterns
- DOM manipulation and event handling
- API integration and asynchronous programming
- Modern development tools and workflows
- React fundamentals and component architecture
- Testing, debugging, and deployment strategies

**Course Structure:** 8 modules, 17 hands-on labs, 1 final project

# What Makes a Frontend Developer?

---

## Core Responsibilities:

- **User Interface (UI):** Visual design and layout
- **User Experience (UX):** Interaction and usability
- **Performance:** Fast loading and responsive applications
- **Accessibility:** Inclusive design for all users
- **Cross-browser Compatibility:** Works everywhere

## Skills You'll Develop:

- Technical proficiency in HTML, CSS, JavaScript
- Problem-solving and debugging
- Design principles and user-centered thinking
- Version control and collaboration
- Modern toolchain and workflow management

# The Modern Web Platform

---

## Evolution of Web Development:

### Then (Early 2000s):

- Static HTML pages
- Table-based layouts
- Inline styles and scripts
- Browser compatibility nightmares

### Now (2024):

- Component-based architectures
- Mobile-first responsive design
- Modern JavaScript with ES6+ features
- Build tools and development workflows
- Progressive Web Applications (PWAs)

**Key Principles:** Semantic HTML, Separation of concerns, Progressive enhancement

# Web Standards and Browser Evolution

---

## Modern Web Standards:

- **HTML5:** Semantic elements, multimedia, APIs
- **CSS3:** Flexbox, Grid, animations, responsive design
- **ES6+:** Modern JavaScript features and syntax
- **Web APIs:** Geolocation, Storage, Canvas, Service Workers

## Browser Capabilities Today:

- Native support for modern JavaScript
- Advanced CSS layout systems
- Built-in developer tools
- Performance optimization features
- Security enhancements (HTTPS, CSP)

# The Frontend Ecosystem

---

## Development Tools:

- **Code Editors:** VS Code, WebStorm, Sublime Text
- **Version Control:** Git, GitHub, GitLab
- **Package Managers:** npm, yarn, pnpm
- **Build Tools:** Vite, webpack, Parcel, Rollup

## Frameworks and Libraries:

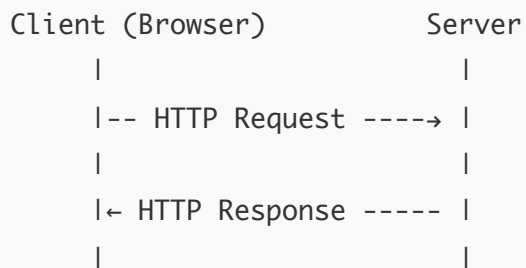
- **React:** Component-based UI library
- **Vue:** Progressive framework
- **Angular:** Full-featured framework
- **Svelte:** Compile-time optimized



# Client-Server Architecture

---

## How Web Applications Work:



### Client-Side (Frontend):

- HTML structure and content
- CSS styling and layout
- JavaScript interactivity and logic
- User interface and experience

### Server-Side (Backend):

- Data processing and storage
- Business logic and APIs
- Authentication and security
- Database management

# Frontend vs Backend vs Full-Stack

---

## Frontend Developer:

- User interface and experience
- Client-side logic and interactivity
- Browser compatibility and performance
- Design implementation

## Backend Developer:

- Server-side logic and APIs
- Database design and management
- Security and authentication
- Infrastructure and deployment

## Full-Stack Developer:

- Both frontend and backend skills
- End-to-end application development
- System architecture decisions
- DevOps and deployment workflows

# Understanding Web Protocols

---

**Internet Protocols enable web communication:**

**TCP/IP** - Transmission Control Protocol/Internet Protocol

- The Internet is a packet-switched network
- TCP collects and reassembles packets
- IP ensures packets reach the right destination

**DNS** - Domain Name System

- Converts between IP addresses and Domain Names
- Example: google.com → 142.250.191.14

**HTTP/HTTPS** - Hypertext Transfer Protocol

- Application-level protocol for web communication
- HTTPS adds security with SSL/TLS encryption

# HTTP Methods and Status Codes

---

## Common HTTP Methods:

- **GET:** Retrieve data from server
- **POST:** Send data to server (create)
- **PUT:** Update existing data
- **DELETE:** Remove data
- **PATCH:** Partial update

## Important Status Codes:

- **200:** OK - Request successful
- **404:** Not Found - Resource doesn't exist
- **500:** Internal Server Error
- **401:** Unauthorized access
- **403:** Forbidden access

# Modern Development Environment

---

## Essential Tools for Front-End Development:

### Code Editor: Visual Studio Code

- Syntax highlighting, IntelliSense, extensions
- Integrated terminal and Git support

### Runtime: Node.js and npm

- JavaScript runtime outside the browser
- Package manager for dependencies

### Build Tool: Vite

- Fast development server with Hot Module Replacement
- Optimized production builds

### Version Control: Git

- Track changes and collaborate effectively

# Setting Up Your Development Environment

---

## System Requirements:

- **Operating System:** Windows 10+, macOS 10.14+, or Linux
- **RAM:** 8GB minimum, 16GB recommended
- **Storage:** 10GB free space for tools and projects
- **Internet:** Broadband connection for downloads

## Installation Order:

1. **VS Code** - Primary code editor
2. **Git** - Version control system
3. **Node.js** - JavaScript runtime and npm
4. **Chrome** - Development browser with DevTools

# VS Code Extensions for Web Development

---

## Essential Extensions:

- **Live Server:** Local development server
- **Prettier:** Code formatting
- **ESLint:** JavaScript linting
- **Auto Rename Tag:** HTML tag synchronization
- **Bracket Pair Colorizer:** Visual bracket matching

## Helpful Extensions:

- **GitLens:** Enhanced Git integration
- **Thunder Client:** API testing
- **Material Icon Theme:** Better file icons
- **Error Lens:** Inline error display

# Lab 01: Working with the Command Line in VSCode

---

## Learning Objectives:

- Open and use integrated terminal in VSCode
- Practice basic command line navigation
- Build confidence with commands for Git, npm, and project setup

## Key Commands:

- `pwd` - Show current directory
- `ls` - List files and folders
- `cd` - Change directory
- `mkdir` - Create directory
- `touch` - Create file



## Module 2: Tools and Workflows

# Version Control with Git

---

## Why Version Control Matters:

- Track changes over time
- Collaborate with team members
- Revert to previous versions
- Branch and merge features

## Git Workflow:

1. `git init` - Initialize repository
2. `git add` - Stage changes
3. `git commit` - Save changes
4. `git push` - Upload to remote repository

# Git Branches and Collaboration

---

## Why Use Branches?

- **Feature Development:** Isolate new features
- **Bug Fixes:** Separate fixes from main code
- **Experimentation:** Try new approaches safely
- **Team Collaboration:** Multiple developers working simultaneously

## Branch Commands:

```
git branch feature-login    # Create new branch
git checkout feature-login  # Switch to branch
git merge feature-login     # Merge branch
git branch -d feature-login # Delete branch
```

# Git Best Practices

---

## Commit Message Guidelines:

- **Clear and Concise:** Describe what was changed
- **Present Tense:** "Add feature" not "Added feature"
- **Limit Length:** 50 characters for summary line
- **Be Specific:** "Fix login button styling" vs "Fix bug"

## Repository Structure:

- **README.md:** Project documentation
- **.gitignore:** Files to exclude from tracking
- **Consistent Naming:** Use clear file and folder names
- **Organize Code:** Logical folder structure

# Package Management with npm

---

## What is npm?

- Node Package Manager
- Manages dependencies for JavaScript projects
- Provides scripts for common tasks

## Key npm Commands:

- `npm init` - Initialize project
- `npm install` - Install dependencies
- `npm run` - Execute scripts
- `npm update` - Update packages

**package.json** - Project configuration file

# Understanding package.json

---

## Essential Fields:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A modern web project",
  "main": "index.js",
  "scripts": {
    "start": "vite",
    "build": "vite build",
    "test": "jest"
  },
  "dependencies": {},
  "devDependencies": {}
}
```

# Dependencies vs DevDependencies

---

**Dependencies:** Needed in production

- Libraries your app needs to run
- React, lodash, axios
- Installed with `npm install package-name`

**DevDependencies:** Development tools only

- Build tools, testing frameworks, linters
- Vite, Jest, ESLint, Prettier
- Installed with `npm install --save-dev package-name`

**Example:**

```
npm install react          # Production dependency
npm install -D vite        # Development dependency
```

# npm Scripts and Automation

---

## Common Script Patterns:

```
{
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "test": "jest",
    "lint": "eslint src/",
    "format": "prettier --write src/"
  }
}
```

## Running Scripts:

```
npm run dev      # Start development server
npm run build    # Build for production
npm test         # Run tests
```



# Browser Developer Tools

---

## Chrome DevTools Features:

### Elements Panel:

- Inspect and modify HTML/CSS live
- Debug layout issues

### Console Panel:

- View JavaScript errors and logs
- Test code interactively

### Sources Panel:

- Set breakpoints and debug JavaScript
- Step through code execution

### Network Panel:

- Monitor HTTP requests and responses

# Lab 02: Using Visual Studio Code Basics

---

## Learning Objectives:

- Master VSCode features and extensions
- Learn Markdown for documentation
- Use Emmet for faster HTML writing
- Customize development environment

## Key VSCode Features:

- Command Palette (Cmd/Ctrl + Shift + P)
- Multi-cursor editing
- Live Server extension
- Markdown preview

## Module 3: HTML Fundamentals

# What is HTML?

---

**HTML = HyperText Markup Language**

- **HyperText** - Text with links to other text
- **Markup** - Tags that describe content structure
- **Language** - Set of rules and syntax

**HTML's Purpose:**

- **Structure content** - Not appearance (that's CSS)
- **Describe meaning** - What content is, not how it looks
- **Create relationships** - Between different pieces of content
- **Enable accessibility** - Screen readers and other tools

**HTML is the skeleton of every web page!**

# HTML Tags, Elements, and Attributes

---

## Understanding the Building Blocks:

**Tag:** The markup syntax

```
<tagname></tagname>
```

**Element:** Complete structure (opening tag + content + closing tag)

```
<h1>This is a heading element</h1>
```

**Attribute:** Extra information about an element

```
 ↑ ↑ attribute name attribute value
```

# HTML Tag Syntax

---

## Opening and Closing Tags:

```
<h1>This is a heading</h1>
<p>This is a paragraph</p>
<strong>This text is important</strong>
```

## Self-Closing Tags (no content):

```

<br />
<hr />
<input type="text" name="username" />
```

## Nested Elements:

```
<p>This paragraph has <strong>bold text</strong> inside it.</p>
<div>
  <h2>A heading inside a div</h2>
  <p>A paragraph inside the same div</p>
</div>
```

# Common HTML Attributes

---

## Universal Attributes (work on most elements):

```
<!-- ID - unique identifier -->
<div id="main-content">Content here</div>

<!-- Class - group elements for styling -->
<p class="highlight important">Special paragraph</p>

<!-- Title - tooltip text -->
<span title="This appears on hover">Hover over me</span>

<!-- Style - inline CSS (avoid when possible) -->
<p style="color: red;">Red text</p>
```

## Element-Specific Attributes:

```
<!-- Links -->
<a href="https://example.com" target="_blank">Visit site</a>

<!-- Images -->


<!-- Form inputs -->
<input type="email" name="email" required placeholder="Enter email" />
```

# Basic HTML Document Structure

---

Every HTML document needs this structure:

```
<!DOCTYPE html>
<!-- Declares HTML5 -->
<html lang="en">
  <!-- Root element with language -->
  <head>
    <!-- Information ABOUT the page -->
    <meta charset="UTF-8" />
    <!-- Character encoding -->
    <title>Page Title</title>
    <!-- Shows in browser tab -->
  </head>
  <body>
    <!-- Visible content goes here -->
    <h1>Main Heading</h1>
    <p>Your content here</p>
  </body>
</html>
```

## Key Parts:

- `<!DOCTYPE html>` - Tells browser this is HTML5
- `<html>` - Root container for entire page
- `<head>` - Metadata (not visible on page)
- `<body>` - All visible content



# Essential HTML Elements

---

## Text Content:

```
<h1>Main Heading</h1>
<!-- Most important heading -->
<h2>Subheading</h2>
<!-- Secondary heading -->
<h3>Sub-subheading</h3>
<!-- And so on... h1-h6 -->

<p>This is a paragraph of text.</p>

<strong>Important text</strong>
<!-- Bold, semantically important -->
<em>Emphasized text</em>
<!-- Italic, semantically emphasized -->

<br />
<!-- Line break -->
<hr />
<!-- Horizontal rule/divider -->
```

## Lists:

```
<!-- Unordered list (bullets) -->
<ul>
  <li>First item</li>
  <li>Second item</li>
</ul>

<!-- Ordered list (numbers) -->
<ol>
  <li>Step one</li>
  <li>Step two</li>
</ol>
```

# Links and Images

---

## Creating Links:

```
<!-- Link to another website -->
<a href="https://www.google.com">Visit Google</a>

<!-- Link to another page on your site -->
<a href="about.html">About Us</a>

<!-- Link to section on same page -->
<a href="#contact">Go to Contact Section</a>

<!-- Email link -->
<a href="mailto:someone@example.com">Send Email</a>
```

## Adding Images:

```
<!-- Basic image -->


<!-- Image with size and linking -->
<a href="large-photo.jpg">
  
</a>
```

**Alt text is crucial for accessibility!**

# HTML5 Semantic Elements

---

Modern HTML5 provides meaningful structure:

## Document Structure:

- `<header>` - Page or section header
- `<nav>` - Navigation links
- `<main>` - Primary content
- `<footer>` - Page or section footer

## Content Organization:

- `<article>` - Self-contained content
- `<section>` - Logical document divisions
- `<aside>` - Sidebar or tangential content
- `<figure>` & `<figcaption>` - Images with captions

**Benefits:** Better SEO, accessibility, and maintainability

# Complete HTML Document Structure

---

Modern HTML5 Template with Semantic Elements:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>My Website - Home</title>
    <meta name="description" content="A brief description of the page" />
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
    <header>
      <nav><!-- Navigation menu --></nav>
    </header>

    <main>
      <article>
        <h1>Main Article Title</h1>
        <p>Article content goes here...</p>
      </article>

      <aside>
        <h2>Related Links</h2>
        <!-- Sidebar content -->
      </aside>
    </main>

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

    <script src="script.js"></script>
  </body>
</html>
```

# HTML Head Section Essentials

---

## Critical Meta Tags:

```
<head>
  <!-- Character encoding (ALWAYS first) -->
  <meta charset="UTF-8" />

  <!-- Responsive design viewport -->
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <!-- Page title (shows in browser tab) -->
  <title>Specific Page Title - Site Name</title>

  <!-- SEO description -->
  <meta name="description" content="Brief description for search engines" />

  <!-- Favicon -->
  <link rel="icon" href="favicon.ico" type="image/x-icon" />

  <!-- CSS stylesheet -->
  <link rel="stylesheet" href="styles.css" />
</head>
```

# HTML Tables

---

## When to Use Tables:

- **Tabular data** (spreadsheet-like information)
- **NOT for layout** (use CSS Grid/Flexbox instead)

## Basic Table Structure:

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>City</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>John Doe</td>
      <td>30</td>
      <td>New York</td>
    </tr>
    <tr>
      <td>Jane Smith</td>
      <td>25</td>
      <td>Los Angeles</td>
    </tr>
  </tbody>
</table>
```

# HTML Forms Basics

---

## Form Structure:

```
<form action="/submit" method="POST">
  <!-- Form fields go here -->

  <button type="submit">Submit Form</button>
</form>
```

## Common Input Types:

```
<!-- Text input -->
<label for="name">Name:</label>
<input type="text" id="name" name="name" required />

<!-- Email input (with validation) -->
<label for="email">Email:</label>
<input type="email" id="email" name="email" required />

<!-- Password input -->
<label for="password">Password:</label>
<input type="password" id="password" name="password" required />

<!-- Number input -->
<label for="age">Age:</label>
<input type="number" id="age" name="age" min="1" max="120" />
```

# Form Elements and Validation

---

## More Input Types:

```
<!-- Textarea for longer text -->
<label for="message">Message:</label>
<textarea id="message" name="message" rows="4" cols="50"></textarea>

<!-- Select dropdown -->
<label for="country">Country:</label>
<select id="country" name="country">
  <option value="">Choose a country</option>
  <option value="us">United States</option>
  <option value="ca">Canada</option>
  <option value="uk">United Kingdom</option>
</select>

<!-- Radio buttons (choose one) -->
<fieldset>
  <legend>Preferred Contact Method:</legend>
  <input type="radio" id="contact-email" name="contact" value="email" />
  <label for="contact-email">Email</label>

  <input type="radio" id="contact-phone" name="contact" value="phone" />
  <label for="contact-phone">Phone</label>
</fieldset>
```



# HTML Validation Attributes

---

## Built-in Form Validation:

```
<!-- Required field -->
<input type="text" name="username" required />

<!-- Minimum/Maximum length -->
<input type="password" name="password" minlength="8" maxlength="20" required />

<!-- Pattern matching (regex) -->
<input
  type="text"
  name="phone"
  pattern="[0-9]{3}-[0-9]{3}-[0-9]{4}"
  placeholder="123-456-7890"
/>

<!-- Email validation -->
<input type="email" name="email" required />

<!-- URL validation -->
<input type="url" name="website" placeholder="https://example.com" />

<!-- Number ranges -->
<input type="number" name="quantity" min="1" max="10" value="1" />

<!-- Date input -->
<input type="date" name="birthday" min="1900-01-01" max="2025-12-31" />
```

# HTML Best Practices

---

## Writing Clean HTML:

### 1. Use Semantic Elements:

```
<!-- ❌ Non-semantic -->
<div class="header">
  <div class="nav">Navigation</div>
</div>

<!-- ✅ Semantic -->
<header>
  <nav>Navigation</nav>
</header>
```

### 2. Always Use Alt Text:

```
<!-- ❌ Missing alt -->


<!-- ✅ Descriptive alt -->

```

### 3. Proper Nesting and Indentation:

```
<!-- ✅ Well-structured -->
<article>
  <h1>Article Title</h1>
  <p>First paragraph with <strong>important text</strong>.</p>
  <p>Second paragraph.</p>
</article>
```

# HTML Accessibility Fundamentals

---

**Making HTML Accessible:**

**Use Proper Headings Hierarchy:**

```
<h1>Main Page Title</h1>
<h2>Section Title</h2>
<h3>Subsection Title</h3>
<h3>Another Subsection</h3>
<h2>Another Section</h2>
```

**Label Form Elements:**

```
<!-- ✅ Proper labeling -->
<label for="email">Email Address:</label>
<input type="email" id="email" name="email" required />

<!-- ✅ Alternative with aria-label -->
<input type="search" aria-label="Search products" placeholder="Search..." />
```

**Use ARIA Attributes When Needed:**

```
<button aria-expanded="false" aria-controls="menu">Menu</button>
<ul id="menu" aria-hidden="true">
  <li><a href="#home">Home</a></li>
  <li><a href="#about">About</a></li>
</ul>
```

# Semantic HTML Benefits

---

## For Search Engines (SEO):

- Better content understanding
- Improved search rankings
- Rich snippets in search results
- Structured data compatibility

## For Accessibility:

- Screen reader navigation
- Keyboard navigation support
- Clear content hierarchy
- Assistive technology compatibility

## For Developers:

- Self-documenting code
- Easier maintenance
- Better team collaboration
- Future-proof markup

# HTML Attributes and Properties

---

## Global Attributes (work on any element):

- `id` - Unique identifier
- `class` - CSS styling hook
- `data-*` - Custom data attributes
- `title` - Tooltip text
- `lang` - Language specification
- `hidden` - Hide element

## Example:

```
<div
  id="main-content"
  class="container active"
  data-theme="dark"
  title="Main content area"
>
  Content here
</div>
```

# HTML Forms and User Input

---

## Essential Form Elements:

### Input Types:

- `<input type="text">` - Text fields
- `<input type="email">` - Email validation
- `<input type="password">` - Hidden input
- `<input type="number">` - Numeric input
- `<input type="date">` - Date picker

### Form Structure:

```
<form action="/submit" method="POST">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" required />
  <button type="submit">Submit</button>
</form>
```

# Form Validation and Accessibility

---

## HTML5 Validation Attributes:

```
<input
  type="email"
  required
  pattern="[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$"
  minlength="5"
  maxlength="100"
  placeholder="Enter your email"
/>
```

## Accessibility Best Practices:

- Always use `<label>` elements
- Associate labels with inputs using `for` attribute
- Provide helpful error messages
- Use fieldsets for grouping related inputs
- Include instructions and requirements

# Advanced Form Elements

---

## Selection Elements:

```
<!-- Dropdown -->
<select name="country">
  <option value="us">United States</option>
  <option value="ca">Canada</option>
</select>

<!-- Radio buttons -->
<input type="radio" id="small" name="size" value="small" />
<label for="small">Small</label>

<!-- Checkboxes -->
<input type="checkbox" id="newsletter" name="newsletter" />
<label for="newsletter">Subscribe to newsletter</label>

<!-- Text area -->
<textarea name="message" rows="4" cols="50"></textarea>
```



# Lab 03: Controlling Your Versions with Git

---

## Learning Objectives:

- Initialize Git repositories
- Stage and commit changes
- Work with remote repositories
- Understand Git workflow and best practices

## Key Git Commands:

- `git init` - Initialize repository
- `git add` - Stage changes
- `git commit` - Save changes with message
- `git push` - Upload to remote repository

# Lab 04: Initializing npm

---

## Learning Objectives:

- Initialize npm package.json
- Understand package.json structure
- Configure project metadata
- Set up npm scripts

## Key Concepts:

- `npm init` command
- Package.json configuration
- Project dependencies vs devDependencies
- npm script basics

## Module 4: CSS Fundamentals

# Introduction to CSS

---

## What is CSS?

- **Cascading Style Sheets** - Controls presentation and layout
- **Separation of Concerns** - HTML for structure, CSS for styling
- **Cascading** - Styles flow down and can override each other
- **Style Sheets** - Collections of style rules

## Why Use CSS?

- **Consistent Styling** across multiple pages
- **Maintainability** - Change styles in one place
- **Responsive Design** - Adapt to different screen sizes
- **Enhanced User Experience** - Better visual design

# Adding CSS to HTML

---

Three Ways to Include CSS:

## 1. External Stylesheet (Recommended):

```
<head>
  <link rel="stylesheet" href="styles.css" />
</head>
```

## 2. Internal Styles:

```
<head>
  <style>
    h1 {
      color: blue;
    }
  </style>
</head>
```

## 3. Inline Styles (Avoid):

```
<h1 style="color: blue;">Heading</h1>
```

# CSS Syntax and Rules

---

## CSS Rule Structure:

```
selector {  
  property: value;  
  another-property: another-value;  
}
```

## Example CSS Rules:

```
/* Element selector */  
h1 {  
  color: blue;  
  font-size: 24px;  
  margin-bottom: 10px;  
}  
  
/* Class selector */  
.highlight {  
  background-color: yellow;  
  padding: 5px;  
}  
  
/* ID selector */  
#main-title {  
  text-align: center;  
  font-weight: bold;  
}
```

# CSS Selectors Fundamentals

---

## Basic Selector Types:

### Element Selectors:

```
h1 {  
  color: blue;  
}  
p {  
  margin: 10px;  
}  
div {  
  border: 1px solid black;  
}
```

### Class Selectors:

```
.warning {  
  color: red;  
}  
.button {  
  padding: 10px;  
}  
.container {  
  max-width: 1200px;  
}
```

### ID Selectors:

```
#header {  
  background: navy;  
}  
#main-content {  
  padding: 20px;  
}
```

# Advanced CSS Selectors

---

## Combinator Selectors:

```
/* Descendant selector */
.container p {
  margin: 15px;
}

/* Child selector */
.nav > li {
  display: inline-block;
}

/* Adjacent sibling */
h1 + p {
  font-weight: bold;
}

/* General sibling */
h1 ~ p {
  color: gray;
}
```

## Attribute Selectors:

```
input[type='text'] {
  border: 1px solid blue;
}
a[href^='https'] {
  color: green;
}
img[alt] {
  border: 2px solid red;
}
```



# CSS Specificity and Cascade

---

Specificity Hierarchy (highest to lowest):

1. Inline styles `style="..."` (1000 points)
2. IDs `#header` (100 points)
3. Classes, attributes, pseudo-classes `.nav`, `[type]`, `:hover` (10 points)
4. Elements `h1`, `div`, `p` (1 point)

Specificity Examples:

```
/* Specificity: 1 */
p {
  color: black;
}

/* Specificity: 10 */
.text {
  color: blue;
}

/* Specificity: 11 */
p.text {
  color: red;
}

/* Specificity: 100 */
#main {
  color: green;
}
```

# CSS Inheritance and Cascade

---

## Inherited Properties:

- Font properties (font-family, font-size, color)
- Text properties (text-align, line-height)
- List properties (list-style)

## Non-Inherited Properties:

- Box model (margin, padding, border)
- Background properties
- Positioning properties

## Example:

```
body {  
  font-family: Arial, sans-serif; /* Inherited */  
  color: #333; /* Inherited */  
  margin: 0; /* NOT inherited */  
}  
  
p {  
  /* Inherits font-family and color from body */  
  margin: 1em 0; /* Must be set explicitly */  
}
```

## Selector Types (by specificity):

1. Inline styles `style="color: red"`
2. IDs `#header`
3. Classes `.nav-item`
4. Elements `h1`

## Specificity Calculation:

```
/* Specificity: 0,0,1,1 */  
h1.title {  
  color: blue;  
}  
  
/* Specificity: 0,1,0,0 */  
#main {  
  color: red;  
}  
  
/* This wins due to higher specificity */
```

# CSS Box Model Fundamentals

---

## Understanding the Box Model:

Every HTML element is a rectangular box with four areas:

1. **Content** - The actual content (text, images)
2. **Padding** - Space between content and border
3. **Border** - Line around the padding
4. **Margin** - Space outside the border

## Visual Representation:

```
|----- margin -----|
| |----- border -----| | | |
| | |---- padding ----| | |
| | |    CONTENT    | | |
| | |-----| | |
| |-----| |
|-----|
```

# Box Model Properties

---

## Setting Box Model Values:

```
.box {  
  /* Content dimensions */  
  width: 200px;  
  height: 100px;  
  
  /* Padding (inside spacing) */  
  padding: 20px;  
  /* OR specific sides */  
  padding: 10px 15px 20px 25px; /* top right bottom left */  
  
  /* Border */  
  border: 2px solid blue;  
  border-width: 2px;  
  border-style: solid;  
  border-color: blue;  
  
  /* Margin (outside spacing) */  
  margin: 10px;  
  margin: 10px auto; /* top/bottom 10px, left/right auto-center */  
}
```

# Box-Sizing Property

---

## The Problem with Default Box Model:

```
.box1 {  
  width: 200px;  
  padding: 20px;  
  border: 2px solid black;  
  /* Total width = 200 + 20 + 20 + 2 + 2 = 244px */  
}
```

## The Solution - Border-Box:

```
.box2 {  
  box-sizing: border-box;  
  width: 200px;  
  padding: 20px;  
  border: 2px solid black;  
  /* Total width = 200px (includes padding and border) */  
}  
  
/* Apply to all elements (recommended) */  
* {  
  box-sizing: border-box;  
}
```

# Margin and Padding Best Practices

---

## Margin Collapse:

```
.element1 {  
  margin-bottom: 20px;  
}  
.element2 {  
  margin-top: 30px;  
}  
/* Actual space between = 30px (not 50px) */
```

## Centering with Margin:

```
.center-block {  
  width: 500px;  
  margin: 0 auto; /* Centers horizontally */  
}
```

## Padding for Internal Spacing:

```
.card {  
  padding: 20px; /* Space inside the card */  
  margin: 20px; /* Space outside the card */  
  border: 1px solid #ccc;  
}
```

# CSS Colors

---

## Color Value Types:

### Named Colors:

```
color: red;  
background-color: blue;  
border-color: green;
```

### Hexadecimal Colors:

```
color: #ff0000; /* Red */  
background: #00ff00; /* Green */  
border: 1px solid #0000ff; /* Blue */  
  
/* Shorthand hex */  
color: #f00; /* Same as #FF0000 */
```

### RGB and RGBA:

```
color: rgb(255, 0, 0); /* Red */  
background: rgba(0, 255, 0, 0.5); /* Semi-transparent green */
```



# CSS Typography Basics

---

## Font Properties:

```
.text-style {  
  /* Font family (fallback fonts) */  
  font-family: 'Arial', 'Helvetica', sans-serif;  
  
  /* Font size */  
  font-size: 18px;  
  font-size: 1.2em; /* Relative to parent */  
  font-size: 1.2rem; /* Relative to root */  
  
  /* Font weight */  
  font-weight: normal; /* 400 */  
  font-weight: bold; /* 700 */  
  font-weight: 300; /* Light */  
  
  /* Font style */  
  font-style: normal;  
  font-style: italic;  
}
```

# Text Styling Properties

---

## Text Properties:

```
.text-formatting {  
  /* Text alignment */  
  text-align: left;  
  text-align: center;  
  text-align: right;  
  text-align: justify;  
  
  /* Text decoration */  
  text-decoration: none;  
  text-decoration: underline;  
  text-decoration: line-through;  
  
  /* Text transform */  
  text-transform: uppercase;  
  text-transform: lowercase;  
  text-transform: capitalize;  
  
  /* Line height (spacing between lines) */  
  line-height: 1.5;  
  line-height: 24px;  
}
```

# Web Fonts

---

## Google Fonts Integration:

### 1. Link in HTML:

```
<head>
  <link
    href="https://fonts.googleapis.com/css2?family=Open+Sans:wght@300;400;700&dis
    rel="stylesheet"
  />
</head>
```

### 2. Use in CSS:

```
body {
  font-family: 'Open Sans', Arial, sans-serif;
}

.heading {
  font-family: 'Open Sans', sans-serif;
  font-weight: 700; /* Bold */
}

.light-text {
  font-weight: 300; /* Light */
}
```

# CSS Positioning Basics

---

## Position Property Values:

### Static (Default):

```
.element {  
  position: static; /* Normal document flow */  
}
```

### Relative:

```
.element {  
  position: relative;  
  top: 10px; /* Move 10px down from normal position */  
  left: 20px; /* Move 20px right from normal position */  
}
```

### Absolute:

```
.element {  
  position: absolute;  
  top: 50px; /* 50px from top of positioned parent */  
  right: 0; /* 0px from right edge */  
}
```

# Advanced Positioning

---

## Fixed Positioning:

```
.header {  
  position: fixed;  
  top: 0;  
  left: 0;  
  width: 100%;  
  background: white;  
  z-index: 1000; /* Stay on top */  
}
```

## Sticky Positioning:

```
.sidebar {  
  position: sticky;  
  top: 20px; /* Stick when 20px from top */  
}
```

## Z-Index (Stacking Order):

```
.modal {  
  position: fixed;  
  z-index: 1000; /* Higher = on top */  
}  
  
.overlay {  
  position: fixed;  
  z-index: 999; /* Behind modal */  
}
```

# Modern CSS Units

---

## Absolute Units:

- `px` - Pixels (screen dots)
- `pt` - Points (print)

## Relative Units:

- `em` - Relative to parent font size
- `rem` - Relative to root font size
- `%` - Percentage of parent
- `vw/vh` - Viewport width/height
- `vmin/vmax` - Smallest/largest viewport dimension

## Best Practices:

- Use `rem` for font sizes
- Use `em` for component spacing
- Use `%` or `vw/vh` for responsive layouts

# Responsive Design Principles

---

## Mobile-First Approach:

```
/* Base styles for mobile */
.container {
  width: 100%;
}

/* Tablet styles */
@media (min-width: 768px) {
  .container {
    width: 750px;
  }
}

/* Desktop styles */
@media (min-width: 1024px) {
  .container {
    width: 1200px;
  }
}
```

## Key Breakpoints:

- Mobile: < 768px
- Tablet: 768px - 1024px
- Desktop: > 1024px

# CSS Flexbox Deep Dive

---

## Flex Container Properties:

```
.container {  
  display: flex;  
  flex-direction: row | column;  
  justify-content: flex-start | center | space-between;  
  align-items: stretch | center | flex-start;  
  flex-wrap: nowrap | wrap;  
  gap: 1rem;  
}
```

## Flex Item Properties:

```
.item {  
  flex-grow: 1; /* How much to grow */  
  flex-shrink: 1; /* How much to shrink */  
  flex-basis: auto; /* Initial size */  
  align-self: auto; /* Override container alignment */  
}
```



# CSS Grid Layout System

---

## Grid Container Setup:

```
.grid-container {  
  display: grid;  
  grid-template-columns: 1fr 2fr 1fr;  
  grid-template-rows: auto 1fr auto;  
  grid-gap: 1rem;  
  grid-template-areas:  
    'header header header'  
    'sidebar main aside'  
    'footer footer footer';  
}
```

## Grid Item Placement:

```
.header {  
  grid-area: header;  
}  
.main {  
  grid-area: main;  
}  
.sidebar {  
  grid-area: sidebar;  
}
```

# CSS Animations and Transitions

---

## Transitions for Smooth Changes:

```
.button {  
  background-color: blue;  
  transition: all 0.3s ease;  
}  
  
.button:hover {  
  background-color: darkblue;  
  transform: scale(1.05);  
}
```

## Keyframe Animations:

```
@keyframes slideIn {  
  from {  
    transform: translateX(-100%);  
  }  
  to {  
    transform: translateX(0);  
  }  
}  
  
.slide-element {  
  animation: slideIn 0.5s ease-out;  
}
```

# Lab 05: Using npm

---

## Learning Objectives:

- Install and manage packages with npm
- Understand node\_modules and package-lock.json
- Use npm scripts for automation
- Work with package versions

## Key npm Commands:

- `npm install` - Install dependencies
- `npm install --save-dev` - Install dev dependencies
- `npm run` - Execute scripts
- `npm update` - Update packages

# Lab 06: Creating a New Project with Vite

---

## Learning Objectives:

- Set up a new project using Vite
- Understand modern build tools
- Configure development environment
- Use hot module replacement (HMR)

## Vite Benefits:

- Fast development server
- Instant hot module replacement
- Optimized production builds
- Modern JavaScript support

## Module 5: JavaScript Fundamentals

# What is JavaScript?

---

## JavaScript = The Programming Language of the Web

- **Dynamic Language** - Code executes at runtime
- **Interpreted** - No compilation step needed
- **Multi-paradigm** - Functional, object-oriented, procedural
- **Event-driven** - Responds to user interactions

## JavaScript Powers:

- **Interactivity** - Respond to clicks, form submissions
- **Dynamic Content** - Update page content without reload
- **Animations** - Smooth transitions and effects
- **API Communication** - Fetch data from servers
- **Modern Apps** - Single Page Applications (SPAs)

JavaScript runs everywhere: **browsers, servers (Node.js), mobile apps!**

# Adding JavaScript to HTML

---

## Three Ways to Include JavaScript:

### 1. External Script File (Recommended):

```
<head>
  <script src="script.js"></script>
</head>
<!-- OR before closing body tag -->
<body>
  <!-- HTML content -->
  <script src="script.js"></script>
</body>
```

### 2. Internal Script:

```
<head>
  <script>
    console.log('Hello from internal script!');
  </script>
</head>
```

### 3. Inline (Avoid):

```
<button onclick="alert('Hello!')">Click me</button>
```

# JavaScript Basics: Variables and Values

---

## What are Variables?

- **Containers** for storing data values
- **Labels** that reference memory locations
- **Dynamic** - can change during program execution

## Creating Variables:

```
// Modern JavaScript (ES6+)
let userName = 'Alice'; // Can be changed
const maxUsers = 100; // Cannot be changed
var oldWay = 'avoid this'; // Old way (function-scoped)

// Changing values
userName = 'Bob'; // ✅ Allowed with let
// maxUsers = 200; // ❌ Error! const cannot change
```

## Variable Naming Rules:

- Must start with letter, underscore, or \$
- Case sensitive ( `userName`  $\neq$  `username` )
- Use camelCase for multiple words



# JavaScript Data Types Fundamentals

---

## Primitive Data Types:

### 1. Numbers:

```
let age = 25; // Integer
let price = 19.99; // Decimal
let negative = -10; // Negative
let infinity = Infinity; // Special number value
```

### 2. Strings (Text):

```
let firstName = 'John'; // Single quotes
let lastName = 'Doe'; // Double quotes
let fullName = `${firstName} ${lastName}`; // Template literal
let multiLine = `Line 1
Line 2`; // Multi-line string
```

### 3. Booleans (True/False):

```
let isLoggedIn = true;
let isComplete = false;
let isValid = age >= 18; // Expression result
```

# More JavaScript Data Types

---

## 4. Special Values:

```
let emptyValue = null; // Intentionally empty
let notAssigned; // undefined (no value assigned)
console.log(notAssigned); // undefined
```

## 5. Arrays (Lists):

```
let colors = ['red', 'green', 'blue'];
let numbers = [1, 2, 3, 4, 5];
let mixed = ['text', 42, true, null];

// Accessing array items (zero-indexed)
console.log(colors[0]); // 'red'
console.log(colors.length); // 3
```

## 6. Objects (Key-Value Pairs):

```
let person = {
  name: 'Alice',
  age: 30,
  isStudent: false,
  address: {
    city: 'New York',
    state: 'NY',
  },
};

// Accessing object properties
console.log(person.name); // 'Alice'
console.log(person['age']); // 30
```

# Basic JavaScript Operators

---

## Arithmetic Operators:

```
let a = 10;
let b = 3;

console.log(a + b); // 13 (addition)
console.log(a - b); // 7 (subtraction)
console.log(a * b); // 30 (multiplication)
console.log(a / b); // 3.333... (division)
console.log(a % b); // 1 (remainder/modulo)
console.log(a ** b); // 1000 (exponentiation)
```

## Assignment Operators:

```
let x = 5; // Basic assignment
x += 3; // x = x + 3; (8)
x -= 2; // x = x - 2; (6)
x *= 2; // x = x * 2; (12)
x /= 3; // x = x / 3; (4)
```

## Comparison Operators:

```
console.log(5 == '5'); // true (loose equality)
console.log(5 === '5'); // false (strict equality)
console.log(5 != '5'); // false (loose inequality)
console.log(5 !== '5'); // true (strict inequality)
console.log(10 > 5); // true
console.log(10 <= 10); // true
```

# JavaScript Control Flow: Conditionals

---

## If Statements:

```
let age = 20;

if (age >= 18) {
  console.log('You are an adult');
} else {
  console.log('You are a minor');
}

// Multiple conditions
if (age < 13) {
  console.log('Child');
} else if (age < 18) {
  console.log('Teen');
} else if (age < 65) {
  console.log('Adult');
} else {
  console.log('Senior');
}
```

## Logical Operators:

```
let isLoggedIn = true;
let isPremium = false;

// AND (&&) - both must be true
if (isLoggedIn && isPremium) {
  console.log('Access premium content');
}

// OR (||) - at least one must be true
if (isLoggedIn || isPremium) {
  console.log('Show some content');
}

// NOT (!) - opposite
if (!isLoggedIn) {
  console.log('Please log in');
}
```

# JavaScript Loops

---

## For Loops:

```
// Traditional for loop
for (let i = 0; i < 5; i++) {
  console.log(`Count: ${i}`);
}

// For...of loop (arrays)
let fruits = ['apple', 'banana', 'orange'];
for (let fruit of fruits) {
  console.log(fruit);
}

// For...in loop (objects)
let person = { name: 'John', age: 30 };
for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

## While Loops:

```
let count = 0;
while (count < 3) {
  console.log(`While count: ${count}`);
  count++;
}

// Do-while (runs at least once)
let input;
do {
  input = prompt('Enter "quit" to stop:');
} while (input !== 'quit');
```

# JavaScript Functions Basics

---

## What are Functions?

- **Reusable blocks of code**
- **Take inputs** (parameters)
- **Return outputs** (return values)
- **Organize code** into logical chunks

## Function Declaration:

```
function greetUser(name) {  
  return `Hello, ${name}!`;  
}  
  
// Calling the function  
let message = greetUser('Alice');  
console.log(message); // "Hello, Alice!"
```

## Function Parameters and Arguments:

```
function calculateArea(width, height) {  
  return width * height;  
}  
  
// Arguments are the actual values passed  
let area = calculateArea(10, 5); // 50  
  
// Default parameters  
function greet(name = 'Guest') {  
  return `Welcome, ${name}!`;  
}
```

# Modern JavaScript (ES6+)

---

## Essential Modern Features:

### Variable Declarations:

```
const name = 'John'; // Immutable
let age = 25; // Block-scoped
// Avoid var          // Function-scoped (legacy)
```

### Arrow Functions:

```
const add = (a, b) => a + b;
const greet = (name) => `Hello, ${name}!`;

// Multiple lines
const processUser = (user) => {
  console.log(`Processing ${user.name}`);
  return user.id;
};
```

### Template Literals:

```
const message = `Welcome, ${name}! You are ${age} years old.`;
const html = `
  <div>
    <h1>${title}</h1>
    <p>${content}</p>
  </div>
`;
```



# JavaScript Data Types

---

## Primitive Types:

```
// Numbers
const age = 25;
const price = 99.99;
const infinity = Infinity;

// Strings
const name = 'John';
const template = `Hello ${name}`;

// Booleans
const isActive = true;
const isComplete = false;

// Special values
const empty = null;
const notDefined = undefined;
```

# Functions in Modern JavaScript

---

## Function Declaration:

```
function greet(name) {  
  return `Hello, ${name}!`;  
}
```

## Function Expression:

```
const greet = function (name) {  
  return `Hello, ${name}!`;  
};
```

## Arrow Function:

```
const greet = (name) => `Hello, ${name}!`;  
  
// Multiple parameters  
const add = (a, b) => a + b;  
  
// Block body for complex logic  
const processUser = (user) => {  
  const formatted = user.name.toUpperCase();  
  return `Welcome, ${formatted}!`;  
};
```

# Scope and Hoisting

---

## Block Scope with let and const:

```
function example() {  
  if (true) {  
    let blockScoped = "I'm only available in this block";  
    const alsoBlockScoped = 'Me too!';  
    var functionScoped = "I'm available in the whole function";  
  }  
  
  // console.log(blockScoped); // Error!  
  console.log(functionScoped); // "I'm available..."  
}
```

## Hoisting Behavior:

```
console.log(x); // undefined (not an error)  
var x = 5;  
  
// console.log(y); // ReferenceError  
let y = 10;
```

# Destructuring and Spread Operator

---

## Array Destructuring:

```
const [first, second, ...rest] = [1, 2, 3, 4, 5];  
// first = 1, second = 2, rest = [3, 4, 5]
```

## Object Destructuring:

```
const { name, age, city = 'Unknown' } = person;
```

## Spread Operator:

```
const newArray = [...oldArray, newItem];  
const newObject = { ...oldObject, newProperty: value };
```

# Conditional Statements and Loops

---

## Modern Conditional Syntax:

```
// Ternary operator
const status = age >= 18 ? 'adult' : 'minor';

// Nullish coalescing
const username = user.name ?? 'Anonymous';

// Optional chaining
const city = user.address?.city ?? 'Unknown';

// Switch with modern syntax
switch (day) {
  case 'monday':
  case 'tuesday':
    return 'weekday';
  case 'saturday':
  case 'sunday':
    return 'weekend';
  default:
    return 'unknown';
}
```

# Modern Loop Patterns

---

## Array Iteration:

```
const numbers = [1, 2, 3, 4, 5];

// for...of loop
for (const num of numbers) {
  console.log(num);
}

// forEach method
numbers.forEach((num) => console.log(num));

// for...in for objects
const person = { name: 'John', age: 30 };
for (const key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

# Error Handling

---

## Try-Catch Blocks:

```
try {
  const data = JSON.parse(jsonString);
  processData(data);
} catch (error) {
  console.error('Parsing failed:', error.message);
} finally {
  console.log('Cleanup code runs regardless');
}
```

## Custom Errors:

```
function validateAge(age) {
  if (age < 0) {
    throw new Error('Age cannot be negative');
  }
  if (age > 150) {
    throw new Error('Age seems unrealistic');
  }
  return true;
}
```

# Lab 07: Using Chrome Developer Tools – Elements Panel

---

## Learning Objectives:

- Navigate the Elements panel in Chrome DevTools
- Inspect and modify HTML elements
- Edit CSS styles in real-time
- Debug layout and styling issues

## Elements Panel Features:

- DOM tree inspection
- Style editing and computed styles
- Box model visualization
- Event listener debugging



# Lab 08: Using Chrome Developer Tools – Sources Panel (JavaScript Debugging)

---

## Learning Objectives:

- Debug JavaScript using the Sources panel
- Set breakpoints and step through code
- Inspect variable values and call stack
- Use console for debugging

## Debugging Features:

- Breakpoint management
- Step over, step into, step out
- Variable inspection and watches
- Call stack analysis

## Module 6: Advanced JavaScript

# Working with Arrays - Fundamentals

---

## Creating and Accessing Arrays:

```
// Creating arrays
let fruits = ['apple', 'banana', 'orange'];
let numbers = new Array(1, 2, 3, 4, 5);
let mixed = ['text', 42, true, { name: 'John' }];

// Accessing elements (zero-indexed)
console.log(fruits[0]); // 'apple'
console.log(fruits[fruits.length - 1]); // 'orange' (last item)

// Checking if it's an array
console.log(Array.isArray(fruits)); // true
```

## Basic Array Properties and Methods:

```
let colors = ['red', 'green', 'blue'];

// Length property
console.log(colors.length); // 3

// Adding elements
colors.push('yellow'); // Add to end
colors.unshift('purple'); // Add to beginning

// Removing elements
let lastColor = colors.pop(); // Remove from end
let firstColor = colors.shift(); // Remove from beginning

// Finding elements
let index = colors.indexOf('green'); // Returns index or -1
let exists = colors.includes('blue'); // true/false
```

# Array Manipulation Methods

---

## Modifying Arrays:

```
let animals = ['cat', 'dog', 'bird', 'fish'];

// Splice - add/remove elements at specific position
animals.splice(2, 1, 'hamster', 'rabbit'); // Remove 1 at index 2, add 2 new
// Result: ['cat', 'dog', 'hamster', 'rabbit', 'fish']

// Slice - create new array from portion
let pets = animals.slice(0, 3); // First 3 elements (doesn't modify original)

// Join - convert to string
let animalString = animals.join(', '); // 'cat, dog, hamster'

// Reverse and sort
animals.reverse(); // Modifies original
animals.sort(); // Alphabetical sort (modifies original)
```

## Array Searching and Testing:

```
let scores = [85, 92, 78, 96, 88];

// Find methods
let highScore = scores.find((score) => score > 90); // 92 (first match)
let highIndex = scores.findIndex((score) => score > 90); // 1 (index of first match)

// Testing methods
let allPassing = scores.every((score) => score >= 70); // true (all pass test)
let someFailing = scores.some((score) => score < 80); // true (at least one passed)
```

# Working with Objects - Fundamentals

---

## Creating and Accessing Objects:

```
// Object literal syntax (most common)
let person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30,
  isEmployed: true,
  address: {
    street: '123 Main St',
    city: 'Anytown',
    state: 'CA',
  },
};

// Accessing properties
console.log(person.firstName); // Dot notation
console.log(person['last-name']); // Bracket notation (for special chars)
console.log(person.address.city); // Nested properties
```

## Adding, Modifying, and Deleting Properties:

```
// Adding new properties
person.email = 'john@example.com';
person['phone'] = '555-1234';

// Modifying existing properties
person.age = 31;
person.isEmployed = false;

// Deleting properties
delete person.phone;

// Checking if property exists
console.log('email' in person); // true
console.log(person.hasOwnProperty('age')); // true
```

# Object Methods and this Keyword

---

## Objects with Methods:

```
let calculator = {
  result: 0,

  add: function (number) {
    this.result += number;
    return this; // Enable method chaining
  },

  subtract: function (number) {
    this.result -= number;
    return this;
  },

  getValue: function () {
    return this.result;
  },

  reset: function () {
    this.result = 0;
    return this;
  },
};

// Using the object methods
calculator.add(10).subtract(3).add(5); // Method chaining
console.log(calculator.getValue()); // 12
```

## Modern Object Methods (ES6+):

```
// Shorthand method syntax
let user = {
  name: 'Alice',

  // Instead of: greet: function() { ... }
  greet() {
    return `Hello, I'm ${this.name}`;
  },

  // Arrow functions don't have their own 'this'
  sayBye: () => {
    // 'this' refers to global object, not 'user'
    return 'Goodbye';
  },
};
```

# Arrays and Objects

---

## Modern Array Methods:

```
const numbers = [1, 2, 3, 4, 5];

// Transformation
const doubled = numbers.map((n) => n * 2);

// Filtering
const evens = numbers.filter((n) => n % 2 === 0);

// Reduction
const sum = numbers.reduce((acc, n) => acc + n, 0);

// Finding
const found = numbers.find((n) => n > 3);
```

## Object Methods:

```
const keys = Object.keys(obj);
const values = Object.values(obj);
const entries = Object.entries(obj);
```



# Advanced Array Methods

---

## Functional Programming Approaches:

```
const users = [  
  { name: 'John', age: 30, active: true },  
  { name: 'Jane', age: 25, active: false },  
  { name: 'Bob', age: 35, active: true },  
];  
  
// Method chaining  
const result = users  
  .filter((user) => user.active)  
  .map((user) => ({ ...user, name: user.name.toUpperCase() })))  
  .sort((a, b) => a.age - b.age);  
  
// Some and every  
const hasActiveUsers = users.some((user) => user.active);  
const allAdults = users.every((user) => user.age >= 18);
```

# Object-Oriented JavaScript

---

## Class Syntax:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    return `Hello, I'm ${this.name}`;
  }

  get isAdult() {
    return this.age >= 18;
  }

  static fromString(str) {
    const [name, age] = str.split(',');
    return new Person(name, parseInt(age));
  }
}

const john = new Person('John', 30);
const jane = Person.fromString('Jane,25');
```

# JavaScript Modules

---

## Exporting from modules:

```
// math.js
export const PI = 3.14159;

export function add(a, b) {
  return a + b;
}

export default function multiply(a, b) {
  return a * b;
}
```

## Importing modules:

```
// main.js
import multiply, { PI, add } from './math.js';
import * as MathUtils from './math.js';

console.log(add(2, 3)); // 5
console.log(multiply(4, 5)); // 20
console.log(MathUtils.PI); // 3.14159
```

# Introduction to the DOM

---

## What is the DOM?

- **Document Object Model** - Programming interface for HTML documents
- **Tree Structure** - Hierarchical representation of HTML elements
- **Live Representation** - Changes update the page in real-time
- **JavaScript Interface** - How JavaScript interacts with HTML/CSS

## DOM Tree Example:

```
document
├── html
│   ├── head
│   │   ├── title
│   │   └── meta
│   └── body
│       ├── header
│       ├── main
│       │   ├── h1
│       │   └── p
│       └── footer
```

# Selecting DOM Elements

---

## Basic Selection Methods:

```
// By ID (returns single element or null)
const header = document.getElementById('main-header');

// By class name (returns HTMLCollection)
const buttons = document.getElementsByClassName('btn');

// By tag name (returns HTMLCollection)
const paragraphs = document.getElementsByTagName('p');

// Modern query selectors (recommended)
const firstButton = document.querySelector('.btn'); // First match
const allButtons = document.querySelectorAll('.btn'); // All matches

// Advanced CSS selectors
const navLinks = document.querySelectorAll('nav a'); // All links in nav
const activeItems = document.querySelectorAll('.item.active'); // Multiple classes
```

## Element Relationships:

```
const element = document.querySelector('.content');

// Parent/child relationships
const parent = element.parentElement;
const children = element.children; // HTMLCollection
const childNodes = element.childNodes; // NodeList (includes text nodes)

// Sibling relationships
const nextSibling = element.nextElementSibling;
const prevSibling = element.previousElementSibling;
```

# Reading and Modifying Content

---

## Text Content:

```
const heading = document.querySelector('h1');

// Get text content (no HTML tags)
console.log(heading.textContent); // "Welcome to My Site"

// Set text content (escapes HTML)
heading.textContent = 'New Heading Text';

// Get/set inner HTML (includes HTML tags)
console.log(heading.innerHTML); // "Welcome <em>to</em> My Site"
heading.innerHTML = 'New <strong>Bold</strong> Heading';
```

## Attributes:

```
const image = document.querySelector('img');

// Get attribute values
const src = image.getAttribute('src');
const alt = image.getAttribute('alt');

// Set attribute values
image.setAttribute('src', 'new-image.jpg');
image.setAttribute('alt', 'New image description');

// Remove attributes
image.removeAttribute('title');

// Common attributes have direct properties
image.src = 'another-image.jpg'; // Same as setAttribute('src', ...)
image.alt = 'Another description';
```

# Modifying Styles and Classes

---

## CSS Classes:

```
const element = document.querySelector('.box');

// Check if class exists
if (element.classList.contains('active')) {
  console.log('Element is active');
}

// Add classes
element.classList.add('highlighted');
element.classList.add('large', 'rounded'); // Multiple classes

// Remove classes
element.classList.remove('old-style');

// Toggle classes (add if not present, remove if present)
element.classList.toggle('visible');

// Replace classes
element.classList.replace('old-class', 'new-class');
```

## Inline Styles:

```
const box = document.querySelector('.box');

// Set individual style properties
box.style.backgroundColor = 'blue';
box.style.width = '200px';
box.style.display = 'none';

// Set multiple styles at once
box.style.cssText = 'background-color: red; width: 300px; height: 200px;';

// Get computed styles (actual applied styles)
const computedStyle = window.getComputedStyle(box);
console.log(computedStyle.backgroundColor); // Actual background color
```

# Creating and Modifying Elements

---

## Creating New Elements:

```
// Create new elements
const newDiv = document.createElement('div');
const newParagraph = document.createElement('p');
const newImage = document.createElement('img');

// Set properties and content
newDiv.textContent = 'This is a new div';
newDiv.className = 'dynamic-content';
newDiv.id = 'generated-div';

newImage.src = 'photo.jpg';
newImage.alt = 'A dynamically created image';

// Create text nodes
const textNode = document.createTextNode('Plain text content');
```

## Adding Elements to the DOM:

```
const container = document.querySelector('.container');
const newElement = document.createElement('p');
newElement.textContent = 'New paragraph';

// Append to end
container.appendChild(newElement);

// Insert at specific position
const firstChild = container.firstChild;
container.insertBefore(newElement, firstChild);

// Modern insertion methods (IE not supported)
container.prepend(newElement); // Add to beginning
container.append(newElement); // Add to end
element.before(newElement); // Insert before element
element.after(newElement); // Insert after element
```



# Removing and Replacing Elements

---

## Removing Elements:

```
const elementToRemove = document.querySelector('.remove-me');

// Modern way (remove itself)
elementToRemove.remove();

// Traditional way (remove from parent)
elementToRemove.parentElement.removeChild(elementToRemove);

// Remove all children
const container = document.querySelector('.container');
container.innerHTML = ''; // Quick but loses event listeners

// Better way to remove all children
while (container.firstChild) {
  container.removeChild(container.firstChild);
}
```

## Replacing Elements:

```
const oldElement = document.querySelector('.old');
const newElement = document.createElement('div');
newElement.textContent = 'Replacement content';

// Replace element
oldElement.parentElement.replaceChild(newElement, oldElement);

// Modern way
oldElement.replaceWith(newElement);
```

# DOM Manipulation

---

## Modern DOM API:

```
// Selection
const element = document.querySelector('.my-class');
const elements = document.querySelectorAll('.item');

// Modification
element.textContent = 'New text';
element.innerHTML = '<strong>Bold text</strong>';
element.classList.add('active');

// Creation
const newElement = document.createElement('div');
newElement.setAttribute('data-id', '123');
parent.appendChild(newElement);
```

# Advanced DOM Techniques

---

## Working with Multiple Elements:

```
// Select all buttons and add event listeners
const buttons = document.querySelectorAll('.btn');
buttons.forEach((button) => {
  button.addEventListener('click', handleClick);
});

// Create elements with attributes
function createCard(title, content) {
  const card = document.createElement('div');
  card.className = 'card';
  card.innerHTML = `
    <h3>${title}</h3>
    <p>${content}</p>
  `;
  return card;
}

// Insert elements at specific positions
const container = document.querySelector('#container');
const newCard = createCard('Title', 'Content');
container.insertBefore(newCard, container.firstChild);
```

# DOM Events Deep Dive

---

## Event Object Properties:

```
function handleEvent(event) {  
  console.log('Event type:', event.type);  
  console.log('Target element:', event.target);  
  console.log('Current target:', event.currentTarget);  
  console.log('Mouse position:', event.clientX, event.clientY);  
  console.log('Key pressed:', event.key);  
  
  // Prevent default behavior  
  event.preventDefault();  
  
  // Stop event propagation  
  event.stopPropagation();  
}
```

# Event Handling

---

## Modern Event Handling:

```
// Event listeners
button.addEventListener('click', handleClick);

// Event object
function handleClick(event) {
  event.preventDefault();
  console.log(event.target);
}

// Event delegation
container.addEventListener('click', (event) => {
  if (event.target.matches('.button')) {
    // Handle button click
  }
});
```

# Lab 09: Creating an HTML Form

---

## Learning Objectives:

- Create interactive forms with proper validation
- Use semantic form elements and attributes
- Implement accessibility best practices
- Handle form data with JavaScript

## Key Concepts:

- Form validation attributes ( `required` , `pattern` )
- Label-input relationships
- Form accessibility
- Modern input types

# Lab 10: Using CSS Selectors

---

## Learning Objectives:

- Master different types of CSS selectors
- Understand selector specificity
- Apply styles effectively
- Practice advanced selector techniques

## Selector Types:

- Element, class, and ID selectors
- Attribute selectors
- Pseudo-classes and pseudo-elements
- Combinator selectors

# Lab 11: Positioning with CSS (and Flexbox)

---

## Learning Objectives:

- Master CSS positioning properties
- Use Flexbox for flexible layouts
- Create responsive design patterns
- Build common UI components

## Key Concepts:

- Static, relative, absolute, fixed positioning
- Flexbox container and item properties
- Alignment and distribution
- Responsive layout techniques



## Module 7: APIs and Asynchronous JavaScript

# Working with APIs

---

## Fetch API for HTTP Requests:

```
// GET request
const response = await fetch('/api/users');
const users = await response.json();

// POST request
const response = await fetch('/api/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(userData),
});
```

## Error Handling:

```
try {
  const data = await fetchUserData();
  console.log(data);
} catch (error) {
  console.error('Failed to fetch data:', error);
}
```

# Asynchronous JavaScript

---

## Promises and Async/Await:

```
// Promise-based
function fetchData() {
  return fetch('/api/data')
    .then((response) => response.json())
    .then((data) => console.log(data))
    .catch((error) => console.error(error));
}

// Async/await
async function fetchData() {
  try {
    const response = await fetch('/api/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

# Understanding Promises

---

## Promise States:

- **Pending:** Initial state, neither fulfilled nor rejected
- **Fulfilled:** Operation completed successfully
- **Rejected:** Operation failed

```
// Creating a Promise
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = Math.random() > 0.5;
    if (success) {
      resolve({ data: 'Success!' });
    } else {
      reject(new Error('Failed to fetch'));
    }
  }, 1000);
});

// Using the Promise
fetchData
  .then((result) => console.log(result))
  .catch((error) => console.error(error));
```

# Async/Await Best Practices

---

## Error Handling with Async/Await:

```
async function fetchUserData(userId) {
  try {
    const userResponse = await fetch(`/api/users/${userId}`);
    if (!userResponse.ok) {
      throw new Error(`HTTP error! status: ${userResponse.status}`);
    }

    const user = await userResponse.json();

    const postsResponse = await fetch(`/api/users/${userId}/posts`);
    const posts = await postsResponse.json();

    return { user, posts };
  } catch (error) {
    console.error('Failed to fetch user data:', error.message);
    throw error; // Re-throw if needed
  }
}
```

# Working with Multiple Promises

---

## Promise.all for Parallel Execution:

```
async function fetchAllData() {
  try {
    const [users, posts, comments] = await Promise.all([
      fetch('/api/users').then((r) => r.json()),
      fetch('/api/posts').then((r) => r.json()),
      fetch('/api/comments').then((r) => r.json()),
    ]);

    return { users, posts, comments };
  } catch (error) {
    console.error('One or more requests failed:', error);
  }
}
```

## Promise.allSettled for Handling Mixed Results:

```
const results = await Promise.allSettled([
  fetch('/api/users'),
  fetch('/api/posts'),
  fetch('/api/comments'),
]);

results.forEach((result, index) => {
  if (result.status === 'fulfilled') {
    console.log(`Request ${index} succeeded:`, result.value);
  } else {
    console.log(`Request ${index} failed:`, result.reason);
  }
});
```

# REST API Conventions

---

## HTTP Methods and Usage:

Method	Purpose	Example
GET	Retrieve data	GET /api/users
POST	Create new resource	POST /api/users
PUT	Update entire resource	PUT /api/users/123
PATCH	Partial update	PATCH /api/users/123
DELETE	Remove resource	DELETE /api/users/123

## Request/Response Structure:

```
// POST request with JSON data
const response = await fetch('/api/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    Authorization: 'Bearer token123',
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com',
  }),
});
```

# Lab 12: Variables, Arrays, and Constants in JavaScript

---

## Learning Objectives:

- Use modern variable declarations ( `const` , `let` )
- Work with arrays and array methods
- Understand scope and hoisting
- Practice data manipulation techniques

## Key Concepts:

- Variable declarations and scope
- Array creation and manipulation
- Const vs let vs var
- Modern JavaScript syntax



# Lab 13: Using Chrome DevTools – JavaScript Console

---

## Learning Objectives:

- Master the JavaScript Console in Chrome DevTools
- Debug JavaScript code effectively
- Test JavaScript expressions interactively
- Analyze runtime errors and warnings

## Console Features:

- Interactive JavaScript execution
- Error and warning messages
- Console logging methods
- Performance monitoring

# Module 8: Modern Frameworks and Deployment

# Introduction to Modern Frameworks

---

## Why Use Frameworks?

- Component-based architecture
- State management
- Virtual DOM for performance
- Rich ecosystem and tooling

## Popular Options:

- **React** - Component-based library
- **Vue** - Progressive framework
- **Angular** - Full-featured framework
- **Svelte** - Compile-time optimization

## Key Concepts:

- Components and props
- State and lifecycle
- Event handling
- Routing and navigation

# Introduction to React

---

## What is React?

- **Library, not framework:** Focused on UI rendering
- **Component-based:** Build encapsulated components
- **Declarative:** Describe what UI should look like
- **Virtual DOM:** Efficient updates and rendering

## React Component Example:

```
function Welcome({ name }) {  
  return <h1>Hello, {name}!</h1>;  
}  
  
function App() {  
  return (  
    <div>  
      <Welcome name="John" />  
      <Welcome name="Jane" />  
    </div>  
  );  
}
```

# JSX and Component Composition

---

## JSX Syntax Rules:

```
function UserCard({ user }) {
  const isOnline = user.status === 'online';

  return (
    <div className="user-card">
      <img src={user.avatar} alt={`${user.name}'s avatar`} />
      <h3>{user.name}</h3>
      <span className={isOnline ? 'online' : 'offline'}>
        {isOnline ? 'Online' : 'Offline'}
      </span>
      {user.isAdmin && <span className="admin-badge">Admin</span>}
    </div>
  );
}
```

## Key JSX Rules:

- Use `className` instead of `class`
- Self-closing tags must end with `/>`
- JavaScript expressions in curly braces `{}`
- Must return single root element (or Fragment)

# React Hooks

---

## useState for State Management:

```
import { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}
```

## useEffect for Side Effects:

```
import { useState, useEffect } from 'react';

function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`/api/users/${userId}`)
      .then((response) => response.json())
      .then(setUser);
  }, [userId]); // Dependency array

  if (!user) return <div>Loading...</div>;

  return <div>Welcome, {user.name}!</div>;
}
```

# Component Communication

---

## Props Flow (Parent to Child):

```
function Parent() {
  const handleChildClick = (message) => {
    alert(`Child says: ${message}`);
  };

  return <Child name="John" onClick={handleChildClick} />;
}

function Child({ name, onClick }) {
  return (
    <div>
      <h3>Hello, {name}!</h3>
      <button onClick={() => onClick('Hello from child!')}>
        Click me
      </button>
    </div>
  );
}
```

# Build Tools and Development Workflow

---

## Modern Build Pipeline:

**Vite** - Next-generation build tool

- Lightning-fast development server
- Hot Module Replacement (HMR)
- Optimized production builds

## Package Management:

- npm/yarn for dependency management
- package.json for project configuration
- Semantic versioning

## Code Quality:

- ESLint for code linting
- Prettier for code formatting
- Git hooks for automated checks



# Modern Development Workflow

---

## Development Process:

1. **Planning:** Requirements and design
2. **Setup:** Initialize project with build tools
3. **Development:** Write code with hot reloading
4. **Testing:** Automated tests and manual QA
5. **Building:** Optimize for production
6. **Deployment:** Deploy to hosting platform
7. **Monitoring:** Track performance and errors

## Continuous Integration/Deployment:

- Automated testing on code changes
- Automated builds and deployments
- Code quality checks and reviews
- Performance monitoring and alerts

# Performance Optimization

---

## Code Splitting:

```
// Dynamic imports for code splitting
const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

## Bundle Analysis:

- Use tools like webpack-bundle-analyzer
- Identify large dependencies
- Remove unused code (tree shaking)
- Optimize images and assets

## Performance Metrics:

- First Contentful Paint (FCP)
- Largest Contentful Paint (LCP)
- Cumulative Layout Shift (CLS)
- First Input Delay (FID)

# Lab 14: Using JavaScript Methods

---

## Learning Objectives:

- Work with JavaScript functions and methods
- Understand method syntax and this binding
- Practice built-in methods for strings and arrays
- Create custom methods for objects

## Method Types:

- String methods (slice, substring, indexOf)
- Array methods (push, pop, shift, unshift)
- Object methods and this context
- Function expressions and arrow functions

# Lab 15: Using JavaScript Objects

---

## Learning Objectives:

- Create and manipulate JavaScript objects
- Understand object properties and methods
- Work with object constructors and prototypes
- Practice object-oriented programming concepts

## Object Concepts:

- Object literal syntax
- Property access and modification
- Object methods and this binding
- Constructor functions and prototypes

# Testing and Debugging

---

## Testing Strategies:

- Unit testing with Jest
- Component testing with React Testing Library
- End-to-end testing with Cypress
- Manual testing and debugging

## Debugging Tools:

- Browser DevTools
- React Developer Tools
- VS Code debugging
- Console logging strategies

## Best Practices:

- Test-driven development
- Continuous integration
- Code coverage metrics

# Lab 16: Performing DOM Manipulation

---

## Learning Objectives:

- Select and modify DOM elements
- Create dynamic content with JavaScript
- Handle user interactions and events
- Build interactive web pages

## DOM Techniques:

- Query selectors and element selection
- Content and attribute manipulation
- Dynamic element creation
- Event handling and delegation

# Deployment and Performance

---

## Deployment Options:

- **Static Hosting:** Netlify, Vercel, GitHub Pages
- **Cloud Platforms:** AWS, Google Cloud, Azure
- **CDN Integration:** Fast global delivery

## Performance Optimization:

- Code splitting and lazy loading
- Image optimization
- Caching strategies
- Bundle size optimization

## Monitoring:

- Performance metrics
- Error tracking
- User analytics

# Lab 17: Building a Movie Review Webpage with jQuery

---

## Learning Objectives:

- Introduction to jQuery library
- Build interactive web interfaces
- Practice DOM manipulation with jQuery
- Create a complete movie review application

## jQuery Features:

- Simplified DOM selection and manipulation
- Event handling with jQuery
- AJAX requests for dynamic content
- Building interactive user interfaces



# Course Summary and Next Steps

---

## What You've Accomplished:

- ✓ Modern HTML5 and semantic markup
- ✓ Advanced CSS with Grid and Flexbox
- ✓ JavaScript ES6+ and modern patterns
- ✓ DOM manipulation and event handling
- ✓ API integration and async programming
- ✓ React fundamentals and component architecture
- ✓ Testing, debugging, and deployment

## Next Steps:

- Build personal projects
- Contribute to open source
- Explore advanced frameworks
- Learn backend development
- Stay updated with web standards

## Resources for Continued Learning:

- MDN Web Docs, JavaScript.info
- React Documentation
- Frontend Masters, freeCodeCamp
- GitHub projects and communities

# Final Project Overview

---

## Capstone Project: Personal Portfolio Website

### Requirements:

- Responsive design with modern CSS
- Interactive features with JavaScript
- API integration for dynamic content
- React components for complex UI
- Professional deployment

### Features to Implement:

- About section with personal information
- Portfolio showcase with project details
- Contact form with validation
- Blog or news section (API-driven)
- Dark/light theme toggle
- Mobile-responsive navigation

### Assessment Criteria:

- Code quality and organization
- User experience and design
- Technical implementation
- Performance and accessibility

# Web Accessibility (a11y)

---

## Core Principles:

- **Perceivable:** Information presented in ways users can perceive
- **Operable:** Interface components must be operable
- **Understandable:** Information and UI operation must be understandable
- **Robust:** Content must be robust enough for various assistive technologies

## Implementation:

```
<!-- Semantic HTML -->
<button aria-label="Close dialog" onclick="closeDialog()">x</button>

<!-- Alt text for images -->


<!-- Form labels -->
<label for="email">Email Address</label>
<input type="email" id="email" required />

<!-- Skip navigation -->
<a href="#main-content" class="skip-link">Skip to main content</a>
```

# Security Best Practices

---

## Frontend Security Concerns:

- **XSS (Cross-Site Scripting):** Sanitize user input
- **CSRF (Cross-Site Request Forgery):** Use CSRF tokens
- **Content Security Policy:** Restrict resource loading
- **HTTPS:** Always use secure connections
- **Input Validation:** Validate on both client and server

## Example CSP Header:

```
<meta
  http-equiv="Content-Security-Policy"
  content="default-src 'self';
          script-src 'self' 'unsafe-inline';
          style-src 'self' 'unsafe-inline';"
/>
```

# Industry Trends and Future

---

## Current Trends:

- **Progressive Web Apps (PWAs):** App-like web experiences
- **JAMstack:** JavaScript, APIs, and Markup
- **Micro-frontends:** Composable frontend architectures
- **WebAssembly:** Near-native performance in browsers
- **AI/ML Integration:** AI-powered user experiences

## Emerging Technologies:

- **Web Components:** Reusable custom elements
- **Server Components:** Server-side rendering with React
- **Edge Computing:** Faster response times globally
- **Web3/Blockchain:** Decentralized applications
- **AR/VR on the Web:** Immersive experiences

# Building Your Portfolio

---

## Essential Portfolio Elements:

- **About Section:** Your story and skills
- **Projects Showcase:** 3-5 quality projects
- **Code Examples:** GitHub repositories
- **Contact Information:** Easy ways to reach you
- **Responsive Design:** Works on all devices

## Project Ideas:

- **Personal Website:** Showcase your skills
- **Todo App:** Demonstrate CRUD operations
- **Weather App:** API integration practice
- **E-commerce Site:** Complex state management
- **Blog Platform:** Full-stack application

# Career Development

---

## Building Your Skills:

- **Practice Regularly:** Code every day
- **Build Projects:** Create real applications
- **Contribute to Open Source:** GitHub contributions
- **Network:** Attend meetups and conferences
- **Stay Updated:** Follow industry blogs and podcasts

## Job Search Tips:

- **Tailor Your Resume:** Match job requirements
- **Prepare for Interviews:** Practice coding challenges
- **Build Your Network:** LinkedIn and community involvement
- **Show Your Work:** Live demos and deployed projects
- **Continuous Learning:** Keep skills current