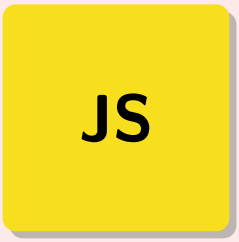
FULL STACK

DEVELOPMENT



JavaScript, Bootstrap, ReactJS & MongoDB

Essential Concepts for Modern Web Development

Author: Rama Bhadra Rao Maddu

Version 1.0 | November 7, 2025

Copyright Notice

Publisher: Self Published
Author: Rama Bhadra Rao Maddu
First Edition: 2024
ISBN: [To be assigned]

This book is intended for educational purposes. While every effort has been made to ensure accuracy, the author assumes no responsibility for errors, omissions, or damages resulting from the use of the information contained herein.

Printed and bound in India.
For more information, contact: [your email/website here]

Book Objectives

This book is designed to equip readers with a solid foundation in both **frontend** and **backend** web development. It provides a gradual learning path starting from the basics of JavaScript and extending to advanced topics like ReactJS and full-stack integration with databases. The overarching aim is to empower learners to independently design, develop, and deploy complete web applications.

General Objectives

By the end of this book, readers will:

- Understand the role of HTML, CSS, and JavaScript as the core building blocks of the web.
- Gain fluency in modern JavaScript, including ES6+ features, DOM manipulation, and asynchronous programming.
- Develop skills in building responsive and mobile-first designs using the Bootstrap framework.
- Learn how to architect interactive and scalable user interfaces with ReactJS.
- Acquire the ability to model, store, and query data efficiently using MongoDB as a NoSQL database.
- Explore how frontend and backend layers integrate to form complete full-stack solutions.

Specific Learning Outcomes

Upon completing this course of study, learners should be able to:

1. Write clean, maintainable JavaScript code using variables, functions, objects, and control structures.
2. Manipulate the Document Object Model (DOM) to create dynamic and interactive web pages.
3. Apply CSS frameworks like Bootstrap to achieve consistent styling and responsive layouts.
4. Implement modular and reusable components in ReactJS, including hooks and state management.
5. Connect frontend applications with backend services and APIs using HTTP and REST principles.
6. Design and interact with MongoDB collections, documents, and queries for persistent storage.

7. Combine all these technologies into a fully functional full-stack project, simulating real-world development workflows.
8. Deploy applications to hosting platforms, making them accessible to end users.

Scope of the Book

This book is intended for:

- Beginners seeking to understand web development fundamentals.
- Intermediate developers aiming to transition into full-stack development.
- Students and professionals preparing for real-world projects, internships, or technical interviews.

In short, the book bridges the gap between theory and practice, enabling readers to progress from basic coding skills to building complete, production-ready web applications.

Contents

CHAPTER 1

1 BASIC JAVASCRIPT

1.1. Introduction to JavaScript

JavaScript has grown from a simple scripting tool into one of the most important programming languages in the world. Originally designed to add small interactive elements to static web pages, it has become a full-fledged language capable of powering complex applications that run across browsers, servers, and even mobile devices. Its flexibility and reach make it a cornerstone of modern software development.

Unlike HTML and CSS, which describe structure and style, JavaScript provides logic and behavior. With it, developers can create dynamic interfaces, respond to user input in real time, and connect applications to remote servers. Because every major browser includes a JavaScript engine, it became the "universal language of the web," accessible to all users without any installation.

Over the years, the language has expanded beyond browsers. With the release of Node.js, JavaScript entered the server world, allowing developers to use the same language for both frontend and backend development. This unification simplified workflows and gave rise to the concept of "full stack" JavaScript, where a single language powers the entire application pipeline.

Today, JavaScript is not just a scripting language but a complete ecosystem. Its annual ECMAScript updates steadily add new features, while frameworks like React, Vue, and Angular dominate the frontend landscape. From small interactive widgets to enterprise-scale platforms, JavaScript remains at the center of digital innovation.

1.1.1 Overview

JavaScript is a high-level, dynamic programming language primarily used to make web pages interactive. It is executed by web browsers and, increasingly, by server runtimes (such as Node.js). Unlike compiled languages, JavaScript is often interpreted or just-in-time compiled by an engine at runtime, which gives it great flexibility and rapid developer feedback. In modern development JavaScript spans multiple domains: client-side user interfaces, server-side APIs, build tooling, desktop and mobile apps, and even embedded devices.

Quick Facts About JavaScript

Created:	May 1995 (10 days)	**Creator:**	Brendan Eich
Original name:	Mocha → LiveScript → JavaScript	**Standard:**	ECMAScript (ECMA-262)
Typical runtimes:	Browsers (V8, SpiderMonkey, JavaScriptCore)	**Also:**	Node.js, Deno, Bun
Common uses:	UI, single-page apps, servers, CLIs	**Paradigms:**	Imperative, OOP, Functional

1.1.2 A Brief History

JavaScript was born at **Netscape Communications** in 1995 during the early days of the commercial web. Netscape wanted a lightweight language that could run directly inside the browser and allow developers to add interactivity to otherwise static HTML pages. Brendan Eich, working at Netscape, created the first implementation in just 10 days. Initially called *Mocha*, then briefly renamed *LiveScript*, it was finally branded *JavaScript*—a marketing decision to ride on the popularity of the Java language, despite the two being unrelated.

Brendan Eich
Creator of JavaScript (1995)
Co-founder of Mozilla

The first standardization effort occurred in **1997**, when ECMA International published **ECMAScript 1**, providing a formal specification that all browser vendors could follow. This was crucial for consistency across competing browsers such as Netscape Navigator and Microsoft Internet Explorer.

Over the next decade, JavaScript's role grew significantly:

- **1999–2004:** ECMAScript 3 introduced regular expressions, better string handling, and try/-catch error handling. It became the foundation for most early JavaScript.
- **2005–2010:** The rise of **AJAX** (Asynchronous JavaScript and XML) revolutionized web applications by enabling asynchronous server communication without reloading the page. Frameworks like jQuery, Prototype, and Dojo emerged to smooth over browser inconsistencies and provide higher-level abstractions.
- **2009: Node.js** was released by Ryan Dahl, using Google's V8 engine. This brought JavaScript to the server side, making it possible to write end-to-end applications using a single language.

- **2015 (ES6/ES2015):** A landmark release that modernized JavaScript with `let`/`const`, classes, arrow functions, template literals, modules, promises, and more. This transformed JavaScript into a modern, large-scale application language.
- **2016–present:** The language adopted an *annual release cycle*, introducing incremental improvements such as `async/await` (2017), `BigInt` (2019), `optional chaining` (2020), and many others.

Through these milestones, JavaScript evolved from a "browser scripting toy" into a **general-purpose, full-stack language** that powers everything from single-page applications to server backends and even IoT devices.

1.1.3 ECMAScript: The Standard

The name "JavaScript" refers to real-world implementations, while **ECMAScript** is the official language specification (ECMA-262). The specification is developed by **TC39**, a technical committee composed of representatives from major technology companies (Google, Mozilla, Microsoft, Apple, and others).

The ECMAScript standard defines:

- The **lexical grammar**: rules for tokens, identifiers, keywords, and literals.
- **Built-in objects**: such as `Object`, `Array`, `String`, `Map`, `Set`, and newer additions like `WeakMap`, `WeakSet`, and `BigInt`.
- The **execution model**: variable scoping, hoisting, closures, the event loop, and asynchronous programming primitives like `Promise` and `async/await`.
- The **module system**: initially CommonJS in Node.js and later standardized as ES Modules (ESM).
- Semantics for **error handling** and the `try/catch/finally` mechanism.

Because browsers and runtimes update at different speeds, developers often use **transpilers** such as Babel and bundlers like Webpack or Vite. These tools let developers write modern ECMAScript (e.g., ES2023) and automatically transform it into older code compatible with legacy browsers. This workflow ensures developers can use the latest language features without breaking compatibility.

1.1.4 JavaScript Engines and Performance

A **JavaScript engine** is the program embedded in a browser (or runtime) that executes JavaScript code. At a high level, engines perform several steps:

1. **Parsing:** Convert source code into an abstract syntax tree (AST).
2. **Interpretation or baseline compilation:** Quickly turn JavaScript into bytecode to start execution fast.
3. **Optimization (JIT):** Frequently executed "hot" code paths are recompiled into highly optimized machine code.
4. **Garbage collection:** Automatic memory management for unused objects.

The major engines in use today are:

- **V8 (Google):** Powers Chrome, Edge, and Node.js. It uses a multi-tier system (Ignition interpreter and TurboFan optimizing compiler) and techniques like hidden classes and inline caching for speed.
- **SpiderMonkey (Mozilla):** The original JS engine (since 1995), now powering Firefox. It includes multiple JIT compilers (Baseline and IonMonkey) and a new optimizing compiler called Warp.
- **JavaScriptCore (Apple):** Also known as Nitro, used in Safari/WebKit. It implements a multi-stage pipeline including the LLInt (low-level interpreter), DFG (data flow graph) JIT, and FTL (faster-than-light) JIT.
- **Chakra (Microsoft):** Formerly used in Internet Explorer and early versions of Edge; now discontinued.

Although engines all follow the ECMAScript specification, each one has its own optimizations. For instance:

- Repeatedly adding properties to objects in different orders can cause "hidden class" deoptimizations in V8.
- Using predictable types (avoiding frequent switching between numbers and strings) helps JIT compilers generate efficient code.
- Avoiding overly complex closures or excessive use of `with` and `eval` leads to better optimization.

Understanding how engines work allows developers to write JavaScript that not only functions correctly but also performs efficiently, particularly for high-performance applications such as games or large-scale web apps.

1.1.5 Client-side vs Server-side Execution

JavaScript runs in two primary environments. The table below compares their key aspects:

Aspect	Client-side (Browser)	Server-side (Node.js)
Execution context	Runs inside the web browser, tied to a webpage/tab.	Runs on the server or local machine as an independent process.
APIs available	DOM, BOM (`window`, `document`), Fetch, Web Storage, Canvas, WebSockets.	File system, HTTP/HTTPS, OS APIs, crypto, streams, worker threads.
Security model	Sandboxed: no direct file system or OS access; CORS restrictions apply.	Full file system and network access (subject to OS permissions).
Typical use cases	UI interactions, form validation, animations, dynamic page updates, single-page apps.	Web servers, REST/GraphQL APIs, CLIs, background jobs, database operations.
Concurrency model	Event-driven, user interactions + async network calls; Web Workers for parallelism.	Event loop with async I/O; worker threads or child processes for CPU-bound work.

Aspect	Client-side (Browser)	Server-side (Node.js)
Global object	`window`, or `globalThis`.	`global`, or `globalThis`.
Module system	ES Modules (ESM), AMD (historical).	CommonJS ('require'), ES Modules (modern Node.js).

1.1.6 Key characteristics

JavaScript is a language with a unique mix of features that influence how programs are written, executed, and reasoned about. Unlike statically typed or class-based languages, JavaScript offers flexibility and dynamism that allow rapid prototyping, but also require careful discipline for building large systems.

- **Dynamic typing:** Variables in JavaScript do not have fixed types; types are associated with values. This allows a single variable to hold different types at different times. While this makes prototyping fast, it can lead to subtle runtime errors if type expectations are not carefully managed.
- **First-class functions:** Functions are treated like any other value: they can be stored in variables, passed as arguments, returned from other functions, and attached as object properties. This capability makes JavaScript especially powerful for functional programming patterns such as higher-order functions, callbacks, and closures.
- **Prototype-based inheritance:** Instead of class-based inheritance (as in Java or C++), JavaScript objects inherit directly from other objects via prototypes. This system is flexible and allows dynamic extension of objects, though modern ECMAScript also provides `class` syntax as a more familiar abstraction over the prototype chain.
- **Event-driven concurrency model:** JavaScript in both browsers and Node.js is single-threaded at its core. Concurrency is achieved through an event loop that schedules callbacks in response to events (user input, timers, network responses). This non-blocking model is efficient for I/O-bound tasks but requires developers to master asynchronous programming techniques such as promises and async/await.
- **Flexible syntax and paradigms:** JavaScript supports imperative (step-by-step instructions), object-oriented (objects and encapsulation), and functional (pure functions, immutability, higher-order functions) programming styles. Modern developers often combine these styles depending on project needs.

Overall, these characteristics make JavaScript adaptable to a wide range of domains, but also demand a strong understanding of asynchronous flow and language quirks.

1.1.7 The modern ecosystem

Beyond the core language, the JavaScript ecosystem has grown into one of the largest and fastest-moving in the software world. This ecosystem spans frontend development, server runtimes, desktop and mobile application frameworks, build tools, and package registries. Developers typically assemble toolchains that combine several of these categories.

- **Frontend frameworks:** Modern user interfaces are commonly built with component-based

frameworks such as **React** (Meta), **Vue** (community-driven), and **Angular** (Google). These frameworks manage UI state, encourage modularity, and provide patterns for building scalable, reactive applications. Each has its own philosophy — React emphasizes a lightweight, declarative core with a large ecosystem; Angular provides a full-featured opinionated framework; Vue aims for simplicity and progressive adoption.

- **Runtimes:** JavaScript is no longer limited to browsers. **Node.js**, introduced in 2009, popularized server-side JavaScript, with a large ecosystem of npm packages. More recent runtimes like **Deno** (created by Ryan Dahl) and **Bun** (built for performance) explore modern defaults such as built-in TypeScript, secure-by-default execution, and faster startup times.

Ryan Dahl
Creator of Node.js (2009)
Also initiated the Deno runtime

- **Tooling:** As applications grew more complex, so did the need for tooling. Bundlers and dev servers such as **Webpack** and **Vite** help manage module resolution, code splitting, and live reloading. Linters (**ESLint**) enforce code quality rules, while formatters (**Prettier**) standardize code style. Testing frameworks like **Jest**, **Mocha**, and **Vitest** provide infrastructure for unit, integration, and end-to-end testing.
- **Package managers:** Libraries are distributed via registries such as the **npm registry**, with clients including **npm**, **yarn**, and **pnpm**. These tools manage dependency graphs, handle semantic versioning, and integrate with continuous integration and deployment workflows.

This ecosystem is vibrant but can also feel overwhelming to newcomers. Choosing tools that balance productivity, performance, and maintainability is a key part of a JavaScript developer's role.

1.1.8 Important contributors and projects

The evolution of JavaScript was driven by individuals, committees, and flagship projects that defined its direction. Understanding these contributors helps place the language in its historical and institutional context.

- **Brendan Eich (1995):** Implemented the first version of JavaScript (then called Mocha) at Netscape in just ten days. Later co-founded Mozilla and contributed to the open web ecosystem. Eich is considered the "father of JavaScript."
- **Ryan Dahl (2009):** Created **Node.js**, bringing JavaScript to the server. Node's design — especially its event-driven, non-blocking I/O model — redefined how developers approached scalable web servers. Dahl later revisited his original design decisions with **Deno**, aiming for stronger security and modern defaults.
- **TC39:** The technical committee (part of ECMA International) responsible for evolving the

ECMAScript standard. Members include representatives from major tech companies (Google, Mozilla, Apple, Microsoft, and others). TC39 operates on a proposal staging process (stages 0–4), allowing the community to track and experiment with upcoming features such as optional chaining, nullish coalescing, and pattern matching.

- **Major projects:** The success of JavaScript is tightly coupled with the engines that run it: Google's **V8**, Mozilla's **SpiderMonkey**, and Apple's **JavaScriptCore** all compete and cooperate in implementing the ECMAScript standard with performance optimizations. On the runtime side, **Node.js** and its successor projects (Deno, Bun) continue to broaden where JavaScript can be applied.

Through these individuals, committees, and projects, JavaScript evolved from a quick scripting experiment into a first-class, multi-platform programming language at the heart of the modern web.

1.1.9 Why learn JavaScript?

JavaScript continues to be one of the most valuable skills for modern developers. Its influence goes far beyond simple browser scripting and now touches almost every domain of software development. Some key reasons to learn JavaScript include:

- **Universal browser support:** JavaScript is the only programming language natively supported in all major web browsers. This makes it the backbone of client-side interactivity, allowing developers to create dynamic websites without additional plugins.
- **Full-stack development:** With the rise of runtimes like Node.js, JavaScript is no longer limited to the browser. Developers can now use the same language for both frontend and backend, simplifying the workflow, reducing context-switching, and allowing code reuse across tiers.
- **Vast ecosystem and community:** JavaScript has the largest package ecosystem in the world (npm registry), containing libraries and frameworks for virtually every problem — from UI rendering and state management to server frameworks and machine learning toolkits. This active community means fast innovation and abundant learning resources.
- **Language evolution and modernization:** The ECMAScript standard now releases yearly updates, steadily improving the language. Features such as `let`/`const`, arrow functions, promises, async/await, modules, and optional chaining have addressed long-standing pain points, making JavaScript more robust, expressive, and maintainable.

Together, these factors make JavaScript not just a language of the web, but a versatile platform for solving real-world problems across many environments.

1.1.10 Further reading and next steps

This introductory chapter laid the groundwork by explaining JavaScript's origins, characteristics, and ecosystem. The next step is to dive into the language mechanics and begin writing and experimenting with code. The suggested learning path is as follows:

1. **Core language mechanics:** Begin with the essentials — values, operators, expressions, and statements. Learn how JavaScript treats variables and scope, and understand its dynamic typing model.

2. **Control flow and data structures:** Explore how to make decisions with `if`, `switch`, and loops. Learn the built-in data structures such as arrays, objects, and strings, which form the foundation of all real-world programs.

3. **Functions and closures:** Master function declarations, expressions, and arrow functions. Understand how closures capture variables and how the `this` keyword behaves in different contexts — both crucial for writing modular and reusable code.

4. **Asynchronous programming:** JavaScript's single-threaded model relies heavily on async techniques. Learn about callbacks, Promises, async/await, and how the event loop schedules tasks. This knowledge is essential for building responsive UIs and scalable server applications.

By following this roadmap, readers can build a strong practical foundation in JavaScript. Each upcoming chapter in this book will expand on these topics with worked examples, best practices, and exercises, gradually developing the ability to design and implement complete web applications.

CHAPTER 2

2 WEB FOUNDATIONS

2.1. HTML Basics

2.1.1 What HTML Is

HTML (HyperText Markup Language) is the foundation of every web page. Think of it as the skeleton that gives structure to web content.

Key Concepts:

- **Structure:** HTML organizes content into logical sections (headers, paragraphs, lists, etc.).
- **Markup:** Uses tags to define different types of content.
- **Semantic elements:** Tags that describe the meaning of the content (for accessibility and SEO).

Why HTML Matters:

- Creates the basic structure browsers can render.
- Provides accessibility hooks for screen readers and assistive technologies.
- Forms the foundation for CSS styling and JavaScript interaction.

2.1.2 Minimal Page Structure

Every HTML document follows a standard structure. Here is the minimal, modern HTML5 page:

Listing 2.1: Basic HTML5 Document Structure

```
<!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>
</head>
<body>
    <!-- Your content goes here -->
</body>
</html>
```

Notes:

- `<!DOCTYPE html>` tells user agents to use standards mode (HTML5).
- Use semantic tags (e.g. `<main>`, `<article>`, `<nav>`) for better structure and accessibility.

2.1.3 Hello World HTML Example

Listing 2.2: Your First HTML Page

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale
       =1.0"/>
    <title>My First Web Page</title>
</head>
<body>
    <h1>Hello, World!</h1>
    <p>Welcome to web development!</p>

    <h2>What I'm Learning</h2>
    <ul>
        <li>HTML structure</li>
        <li>CSS styling</li>
        <li>JavaScript interaction</li>
    </ul>
</body>
</html>
```

2.2. Complete HTML Tags Reference

Table 2.1: HTML5 Tags Reference - Key attributes and usage examples

Tag	Key Attributes	Description and Example
DOCTYPE	–	Declares HTML5 document type. `<!DOCTYPE html>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
`html`	lang	Root element of HTML document. `<html lang="en"> <!-- content --> </html>`
`head`	–	Container for metadata: title, meta, links, scripts. `<head> <title>My Page</title> <meta charset="UTF-8"> </head>`
`meta`	charset, name, content, viewport	Page metadata. `<meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1"> <meta name="description" content="Page description">`
`title`	–	Page title shown in browser tab. `<title>My Website</title>`
`link`	rel, href, type, media	Links to external resources. `<link rel="stylesheet" href="style.css"> <link rel="icon" href="favicon.ico">`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
script	src, async, defer	External or inline JavaScript. `<script src="app.js" defer></script> <script> console.log('Hello World'); </script>`
style	–	Internal CSS styles in head. `<style> h1 { color: red; } .highlight { background: yellow; } </style>`
body	–	Document body containing main visible content. `<body> <h1>Welcome</h1> <p>Content goes here</p> </body>`
a	href, target, rel, download	Anchor link element. `<a href="page.html">Internal Link</a> <a href="https://example.com" target="_blank" rel="noopener">External Link</a> <a href="file.pdf" download>Download File</a>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
`img`	src, alt, width, height, loading	Image element. Always include alt attribute. `<img src="photo.jpg" alt="Beautiful sunset" width="300" height="200"> <img src="lazy.jpg" alt="Lazy loaded" loading="lazy">`
`div`	id, class, style	Generic container block element. `<div id="main-container" class="wrapper"> <div class="card">Content</div> </div>`
`span`	id, class, style	Generic inline container. `<p>This is <span class="highlight">important</span> text.</p>`
`p`	–	Paragraph text. `<p>This is a paragraph with some text content.</p>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example

h1-h6	–	Heading elements from largest to smallest.
		`<h1>Main Title</h1>`
		`<h2>Section Title</h2>`
		`<h3>Subsection</h3>`
		`<h4>Minor Heading</h4>`
		`<h5>Smaller Heading</h5>`
		`<h6>Smallest Heading</h6>`

ul, ol, li	type, start	Unordered and ordered lists with list items.
		`<ul>`
		`<li>First item</li>`
		`<li>Second item</li>`
		`</ul>`
		`<ol start="5">`
		`<li>Fifth item</li>`
		`<li>Sixth item</li>`
		`</ol>`

table	border, cellpadding, cellspacing	Table container with rows and cells.
		`<table border="1">`
		`<thead>`
		`<tr>`
		`<th>Name</th>`
		`<th>Age</th>`
		`</tr>`
		`</thead>`
		`<tbody>`
		`<tr>`
		`<td>John</td>`
		`<td>25</td>`
		`</tr>`
		`</tbody>`
		`</table>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
`tr`	–	Table row containing cells. `<tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr>`
`td, th`	colspan, rowspan, scope	Table data cell and header cell. `<th scope="col">Header</th><td colspan="2">Spans 2 columns</td><td rowspan="3">Spans 3 rows</td>`
`form`	action, method, enctype, target	Form container for user input. `<form action="/submit" method="post" enctype="multipart/form-data"><input type="text" name="username" required><button type="submit">Submit</button></form>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example

input	type, name, value, placeholder, required	Various input types for form data.
		`<input type="text" name="name" placeholder="Enter name">`
		`<input type="email" name="email" required>`
		`<input type="password" name="pwd" minlength="8">`
		`<input type="number" name="age" min="0" max="120">`
		`<input type="date" name="birthday">`
		`<input type="file" name="upload" accept=".pdf,.jpg">`
		`<input type="checkbox" name="agree" value="yes">`
		`<input type="radio" name="gender" value="male">`
		`<input type="hidden" name="token" value="abc123">`
		`<input type="submit" value="Submit Form">`

button	type, onclick, disabled	Clickable button element.
		`<button type="submit">Submit</button>`
		`<button type="button" onclick="alert('Hello')">Click Me</button>`
		`<button type="reset">Reset Form</button>`
		`<button disabled>Disabled Button</button>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
`select`	name, multiple, size	Dropdown selection menu. `<select name="country"><option value="">Choose country</option><option value="us">United States</option><option value="uk" selected>United Kingdom</option></select><select name="colors" multiple size="3"><option value="red">Red</option><option value="blue">Blue</option><option value="green">Green</option></select>`
`option`	value, selected, disabled	Option within select element. `<option value="1">Option 1</option><option value="2" selected>Default Option</option><option value="3" disabled>Disabled Option</option>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
optgroup	label, disabled	Groups options in select element. `<select name="food"><optgroup label="Fruits"><option value="apple">Apple</option><option value="banana">Banana</option></optgroup><optgroup label="Vegetables"><option value="carrot">Carrot</option></optgroup></select>`
textarea	rows, cols, placeholder, maxlength	Multi-line text input area. `<textarea name="message" rows="4" cols="50" placeholder="Enter your message here ..." maxlength="500"></textarea>`
label	for	Label for form controls. `<label for="email">Email Address:</label><input type="email" id="email" name="email"><label><input type="checkbox" name="terms">I agree to terms</label>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
`fieldset`	disabled	Groups related form controls. `<fieldset> <legend>Personal Information</legend> <label>Name: <input type="text" name="name"></label> <label>Age: <input type="number" name="age"></label> </fieldset>`
`legend`	–	Caption for fieldset element. `<fieldset> <legend>Contact Details</legend> <!-- form controls --> </fieldset>`
`datalist`	–	Predefined options for input elements. `<input list="browsers" name="browser"> <datalist id="browsers"> <option value="Chrome"> <option value="Firefox"> <option value="Safari"> </datalist>`
`canvas`	width, height	2D drawing surface for JavaScript graphics. `<canvas id="myCanvas" width="400" height="300"> Your browser does not support canvas. </canvas>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
video	controls, src, poster, autoplay	Video playback element. `<video controls width="400" poster="thumbnail.jpg"><source src="movie.mp4" type="video/mp4"><source src="movie.webm" type="video/webm">Your browser does not support video.</video>`
audio	controls, src, autoplay, loop	Audio playback element. `<audio controls><source src="audio.mp3" type="audio/mpeg"><source src="audio.ogg" type="audio/ogg">Your browser does not support audio.</audio>`
source	src, type, media	Media source for video/audio elements. `<video controls><source src="movie.mp4" type="video/mp4"><source src="movie.webm" type="video/webm"></video>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
iframe	src, width, height, frameborder	Embeds another HTML document. `<iframe src="https://example.com" width="500" height="300" frameborder="0"></iframe>`
nav	–	Navigation links section. `<nav><ul><li><a href="home.html">Home</a></li><li><a href="about.html">About</a></li><li><a href="contact.html">Contact</a></li></ul></nav>`
header	–	Header section of page or article. `<header><h1>Website Title</h1><nav><a href="home.html">Home</a><a href="about.html">About</a></nav></header>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
footer	–	Footer section of page or article. `<footer> <p>&copy; 2024 Company Name. All rights reserved.</p> <p>Contact: info@example.com</p> </footer>`
main	–	Main content area of page. `<main> <article> <h1>Article Title</h1> <p>Main article content...</p> </article> </main>`
section	–	Thematic grouping of content with heading. `<section> <h2>About Us</h2> <p>Information about our company...</p> </section>`
article	–	Self-contained, reusable content. `<article> <header> <h1>Blog Post Title</h1> <time datetime="2024-01-15">January 15, 2024</time> </header> <p>Article content...</p> </article>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
aside	–	Sidebar or tangentially related content. `<aside><h3>Related Links</h3><ul><li><a href="link1.html">Related Article 1</a></li><li><a href="link2.html">Related Article 2</a></li></ul></aside>`
details	open	Collapsible content disclosure widget. `<details><summary>Click to expand</summary><p>This content is hidden by default and can be expanded by clicking the summary.</p></details>`
summary	–	Summary/caption for details element. `<details><summary>Frequently Asked Questions</summary><p>Answer to common questions ...</p></details>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
dialog	open	Modal dialog box or popup window. `<dialog id="myDialog"><h2>Dialog Title</h2><p>Dialog content goes here.</p><button onclick="document.getElementById('myDialog').close()">Close</button></dialog>`
template	–	Template content not rendered until cloned via JavaScript. `<template id="card-template"><div class="card"><h3>Card Title</h3><p>Card content</p></div></template>`
progress	value, max	Progress indicator bar. `<progress value="70" max="100">70%</progress><progress>Indeterminate progress</progress>`
meter	value, min, max, high, low	Scalar measurement within range. `<meter value="6" min="0" max="10">6 out of 10</meter><meter value="0.6" high="0.9" low="0.2">60%</meter>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example

time	datetime	Date and time information.
		`<time datetime="2024-01-15">January 15, 2024</time>`
		`<time datetime="2024-01-15T14:30:00">January 15, 2024 at 2:30 PM</time>`

mark	–	Highlighted or marked text for reference. `<p>Search for <mark>HTML tags</mark> in the document.</p>`
strong	–	Strong importance, serious urgency. `<p><strong>Warning:</strong> This action cannot be undone.</p>`
em	–	Emphasized text, stress emphasis. `<p>I <em>really</em> need to finish this project.</p>`
small	–	Fine print, legal text, copyright. `<p><small>&copy; 2024 Company Name. All rights reserved.</small></p>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example

sub, sup	–	Subscript and superscript text.
		`<p>Water formula: H<sub>2</sub>O</p>`
		`<p>Einstein's equation: E=mc<sup>2</sup></p>`

code	–	Inline code fragment. `<p>Use the <code>console.log()</code> function to debug.</p>`

pre	–	Preformatted text block.
		`<pre>`
		`function hello() {`
		`  console.log("Hello World!");`
		`}`
		`</pre>`

blockquote	cite	Extended quotation from another source.
		`<blockquote cite="https://example.com/quote">`
		`  <p>The only way to do great work is to love what you do.</p>`
		`  <footer>- Steve Jobs</footer>`
		`</blockquote>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
q	cite	Short inline quotation. `<p>As they say, <q>practice makes perfect</q>.</p>`
cite	–	Title of creative work being referenced. `<p>My favorite book is <cite>To Kill a Mockingbird</cite>.</p>`
abbr	title	Abbreviation or acronym with expansion. `<p><abbr title="HyperText Markup Language">HTML</abbr> is the standard markup language.</p>`
address	–	Contact information for author or owner. `<address> Written by <a href="mailto:john@example.com">John Doe</a>  123 Main Street  Anytown, USA 12345 </address>`
del, ins	cite, datetime	Deleted and inserted text changes. `<p>The price is <del>$50</del> <ins>$40</ins>.</p> <ins datetime="2024-01-15">This text was added on Jan 15.</ins>`

Continued on next page

Table 2.1 continued from previous page

Tag	Key Attributes	Description and Example
br	–	Line break within text content. `<p>First line Second line Third line</p>`
hr	–	Thematic break, horizontal rule separator. `<p>Section one content.</p><hr><p>Section two content.</p>`

2.3. CSS Basics

2.3.1 Purpose of CSS

CSS (Cascading Style Sheets) is the language that defines the presentation (look and feel) of web pages. While HTML provides the *structure and content* (headings, paragraphs, images, links), CSS is responsible for the *styling and layout*. This separation of content (HTML) and presentation (CSS) makes web development more organized, reusable, and professional.

Why We Need CSS:

- Without CSS, web pages would appear as plain text with minimal formatting.
- CSS makes websites visually attractive, improves user experience, and conveys branding through colors, typography, and design.
- It allows consistent styling across multiple pages by reusing one stylesheet.

What CSS Does:

- **Controls appearance:** set *colors*, *fonts*, and *text styles*.
- **Manages spacing:** add *margins*, *padding*, and *line-height*.
- **Creates layouts:** arrange elements using **Flexbox**, **CSS Grid**, or **positioning**.
- **Adds visual effects:** apply shadows, gradients, hover effects, transitions, and animations.
- **Responsive design:** adapt pages for desktops, tablets, and mobile screens with `@media` queries.
- **Accessibility:** enhance readability and usability for all users.

Example: Plain HTML vs Styled with CSS

Listing 2.67: Plain HTML without CSS

Without CSS (Plain HTML)

```
<!DOCTYPE html>
<html>
<body>
    <h1>My Website</h1>
    <p>Welcome to my first website. This is plain HTML.</p>
    <button>Click Me</button>
</body>
</html>
```

Listing 2.68: HTML styled with CSS

With CSS Applied

```
<!DOCTYPE html>
<html>
<head>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f9;
            color: #333;
            text-align: center;
            padding: 50px;
        }
        h1 { color: #2c3e50; }
        p  { font-size: 18px; line-height: 1.5; }
        button {
            background-color: #3498db;
            color: white;
            border: none;
            padding: 10px 20px;
            font-size: 16px;
            border-radius: 5px;
            cursor: pointer;
        }
        button:hover {
            background-color: #2980b9;
        }
    </style>
</head>
<body>
    <h1>My Website</h1>
    <p>Welcome to my first website. Styled with CSS!</p>
    <button>Click Me</button>
```

```
</body>
</html>
```

Difference:

- Plain HTML looks basic, with default black text and unstyled buttons.
- Styled with CSS, the page becomes visually appealing, easier to read, and more interactive.

Key Benefits of CSS:

1. **Separation of concerns:** HTML handles content, CSS handles presentation.
2. **Consistency:** one CSS file can control styling for hundreds of pages.
3. **Efficiency:** easy to update the look of an entire website by editing a single stylesheet.
4. **User experience:** good design improves readability, navigation, and accessibility.
5. **Cross-device adaptability:** responsive CSS ensures the website works on mobiles, tablets, and desktops.

2.4. Types of CSS

CSS can be applied to HTML in three primary ways: **inline**, **internal**, and **external**. Each method has its own use-cases, advantages, and disadvantages. Understanding these is essential for writing clean, maintainable, and scalable code.

2.4.1 1. Inline CSS

Definition: Inline CSS is written directly inside an element's `style` attribute. It affects only that specific element.

Listing 2.69: Inline CSS Example

```
<h1 style="color: blue; font-size: 36px;">Blue Heading</h1>
<p style="background-color: yellow; padding: 10px;">
    This paragraph has a yellow background.
</p>
```

Advantages:

- Quick to apply for testing or one-time changes.
- Does not require separate CSS files.
- Useful for email templates where external CSS may not load.

Disadvantages:

- Mixes content with presentation (violates separation of concerns).
- Hard to maintain for large projects (duplicate styles everywhere).
- Cannot define reusable rules or pseudo-classes like `:hover`.

Use-case: Quick demos, small experiments, or overriding styles in controlled environments (e.g., emails).

2.4.2 2. Internal CSS

Definition: Internal CSS is placed inside a `<style>` block within the `<head>` of an HTML document. It applies styles only to that page.

Listing 2.70: Internal CSS Example

```
<!DOCTYPE html>
<html>
<head>
    <style>
        h1 { color: blue; font-size: 36px; text-align: center; }
        p  { background-color: #f0f0f0; padding: 15px; border-radius: 5
           px; }
        .highlight { background-color: yellow; font-weight: bold; }
    </style>
</head>
<body>
    <h1>Welcome</h1>
    <p class="highlight">This is highlighted text</p>
</body>
</html>
```

Advantages:

- Keeps styles in one place instead of mixing them with HTML.
- Useful for small projects or single-page demos.
- Easy to test without creating extra files.

Disadvantages:

- Styles apply only to that page; no reusability across multiple pages.
- Can make the HTML head section very large in bigger projects.
- Slower page loading if many rules are duplicated across pages.

Use-case: Educational examples, one-page websites, or pages with unique styles.

2.4.3 3. External CSS

Definition: External CSS keeps all styles in a separate `.css` file, linked using the `<link>` element inside the HTML `<head>`. This is the **best practice** for real-world projects.

Listing 2.71: HTML with External CSS Link

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href="styles.css">
```

```
</head>
<body>
    <header class="main-header">My Website</header>
    <nav class="navigation">
        <ul>
            <li><a href="#">Home</a></li>
            <li><a href="#">About</a></li>
        </ul>
    </nav>
</body>
</html>
```

Listing 2.72: External CSS File (styles.css)

```
/* Reset and base styles */
* { box-sizing: border-box; margin:0; padding:0; }

body {
    font-family: Arial, sans-serif;
    color: #333;
    line-height: 1.6;
}

.main-header {
    background: #2c3e50;
    color: white;
    padding: 1rem 0;
    text-align: center;
}

.navigation ul {
    list-style: none;
    display: flex;
    gap: 2rem;
    justify-content: center;
}
```

Advantages:

- Separation of structure (HTML) and presentation (CSS).
- Reusability — one CSS file can style hundreds of pages.
- Faster page loading (CSS cached by browser).
- Easier collaboration in large teams.

Disadvantages:

- Requires multiple files (HTML + CSS).

- Not suitable for very quick demos.

Use-case: All modern, professional web development projects.

2.4.4 Comparison of CSS Types

Type	Where written	Pros	Cons
Inline CSS	Inside element's `style` attribute	Quick, no extra files	Hard to maintain, not reusable
Internal CSS	Inside `<style>` in `<head>`	Styles grouped in one place, good for single page	Not reusable across pages, large heads
External CSS	Separate `.css` file linked with `<link>`	Reusable, cached, best practice	Needs extra file(s), not ideal for quick test

2.5. Box Model Concept

Every HTML element is treated as a rectangular box in CSS. The **CSS Box Model** describes how the size of this box is calculated and how elements are spaced. It consists of:

- **Content**: The actual content (text, images, or other data).
- **Padding**: The space between the content and the border.
- **Border**: The visible line surrounding the padding.
- **Margin**: The space outside the border, separating the element from others.

Listing 2.73: Box Model Visualization

```
.box {
    width: 200px;              /* content width */
    padding: 20px;             /* space inside */
    border: 5px solid #333;    /* border */
    margin: 15px;              /* space outside */
}
```

Note: Use `box-sizing: border-box;` to include padding and border in the element's total width/height.

Listing 2.74: Box Sizing Example

```
* { box-sizing: border-box; }
```

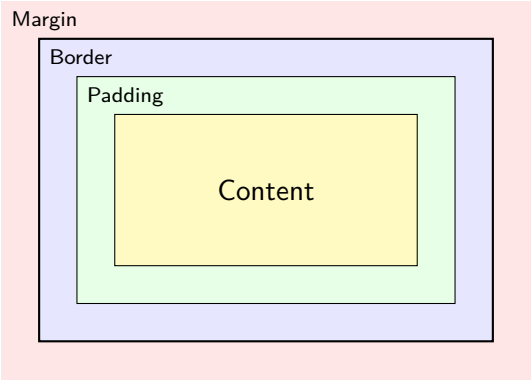


Diagram Explanation:

1. The yellow area is the **Content** (text, images).
2. The green area around it is the **Padding**.
3. The blue ring represents the **Border**.
4. The red area outside the border is the **Margin**.

2.6. comprehensive CSS reference

Table 2.2: CSS Properties Reference - Key values and usage examples

Property	Common Values	Description and Example

`color`	hex, rgb, hsl, named	Sets text color.
		
		Listing 2.75: Text Color Examples
		`h1 { color: #ff0000; }`
		`p { color: rgb(255, 0, 0); }`
		`.text { color: red; }`
		`.highlight { color: hsl(0, 100%, 50%); }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

background-color	hex, rgb, hsl, transparent	Sets background color.
		
		Listing 2.76: Background Color Examples
		`body { background-color: #f0f0f0; }`
		`.card { background-color: white; }`
		`.highlight { background-color : yellow; }`

background-image	url(), gradient functions	Sets background image or gradient.
		
		Listing 2.77: Background Images and Gradients
		`.hero { background-image: url ('hero.jpg'); }`
		`.gradient { background-image: linear-gradient(45deg, blue, red); }`
		`.radial { background-image: radial-gradient(circle, white, black); }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

background-size	cover, contain, px, %	Controls background image size.
		
		Listing 2.78: Background Size Options
		`.cover { background-size: cover; }`
		`.contain { background-size: contain; }`
		`.custom { background-size: 100px 200px; }`
		`.responsive { background-size: 100% auto; }`

background-position	keywords, px, %	Sets background image position.
		
		Listing 2.79: Background Positioning
		`.center { background-position: center; }`
		`.top-right { background-position: top right; }`
		`.custom { background-position: 20px 50px; }`
		`.percent { background-position: 75% 25%; }`

background-repeat	no-repeat, repeat, repeat-x, repeat-y	Controls background repetition.
		
		Listing 2.80: Background Repeat Patterns
		`.no-repeat { background-repeat: no-repeat; }`
		`.repeat-x { background-repeat: repeat-x; }`
		`.tile { background-repeat: repeat; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`font-family`	serif, sans-serif, monospace, font names	Sets font family.
		Listing 2.81: Font Family Definitions
		`body { font-family: Arial, sans-serif; }`
		`h1 { font-family: 'Times New Roman', serif; }`
		`code { font-family: 'Courier New', monospace; }`
		`.custom { font-family: 'Roboto', Helvetica, sans-serif; }`

`font-size`	px, em, rem, %, keywords	Sets font size.
		Listing 2.82: Font Size Examples
		`h1 { font-size: 32px; }`
		`p { font-size: 1rem; }`
		`.large { font-size: 1.5em; }`
		`.small { font-size: 80%; }`
		`.keyword { font-size: large; }`

`font-weight`	normal, bold, 100-900	Sets font thickness.
		Listing 2.83: Font Weight Variations
		`h1 { font-weight: bold; }`
		`.thin { font-weight: 300; }`
		`.normal { font-weight: 400; }`
		`.thick { font-weight: 700; }`
		`.heavy { font-weight: 900; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`font-style`	normal, italic, oblique	Sets font style.
		Listing 2.84: Font Style Options
		`em { font-style: italic; }`
		`.normal { font-style: normal; }`
		`.slanted { font-style: oblique; }`

`text-align`	left, center, right, justify	Aligns text horizontally.
		Listing 2.85: Text Alignment Options
		`.center { text-align: center; }`
		`.right { text-align: right; }`
		`.justify { text-align: justify; }`
		`h1 { text-align: center; }`

`text-decoration`	none, underline, overline, line-through	Adds text decoration.
		Listing 2.86: Text Decoration Styles
		`a { text-decoration: none; }`
		`.underline { text-decoration: underline; }`
		`.strikethrough { text-decoration: line-through; }`
		`.overline { text-decoration: overline; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`text-transform`	none, uppercase, lowercase, capitalize	Transforms text case.
		Listing 2.87: Text Case Transformations
		`.uppercase { text-transform: uppercase; }`
		`.lowercase { text-transform: lowercase; }`
		`.capitalize { text-transform: capitalize; }`
		`.normal { text-transform: none; }`

`line-height`	normal, number, px, em, %	Sets line spacing.
		Listing 2.88: Line Height Examples
		`p { line-height: 1.5; }`
		`.tight { line-height: 1.2; }`
		`.loose { line-height: 2.0; }`
		`.fixed { line-height: 24px; }`

`letter-spacing`	normal, px, em	Sets character spacing.
		Listing 2.89: Letter Spacing Examples
		`.spaced { letter-spacing: 2px; }`
		`.tight { letter-spacing: -0.5px; }`
		`.wide { letter-spacing: 0.1em; }`
		`.normal { letter-spacing: normal; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

word-spacing	normal, px, em	Sets word spacing.
		Listing 2.90: Word Spacing Examples
		`.wide-words { word-spacing: 5 px; }`
		`.tight-words { word-spacing: -2px; }`
		`.em-spacing { word-spacing: 0.2em; }`

width	auto, px, %, vw, em	Sets element width.
		Listing 2.91: Width Property Examples
		`.container { width: 1200px; }`
		`.half { width: 50%; }`
		`.full-viewport { width: 100vw ; }`
		`.auto { width: auto; }`

height	auto, px, %, vh, em	Sets element height.
		Listing 2.92: Height Property Examples
		`.sidebar { height: 400px; }`
		`.full-screen { height: 100vh; }`
		`.half { height: 50%; }`
		`.auto { height: auto; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`max-width`	none, px, %, vw	Sets maximum width constraint.
		
		Listing 2.93: Maximum Width Constraints
		`.container { max-width: 1200 px; }`
		`.responsive { max-width: 100%; }`
		`.constrained { max-width: 60 vw; }`

`max-height`	none, px, %, vh	Sets maximum height constraint.
		
		Listing 2.94: Maximum Height Constraints
		`.modal { max-height: 80vh; }`
		`.scrollable { max-height: 300 px; }`
		`.flexible { max-height: 90%; }`

`min-width`	0, px, %, vw	Sets minimum width constraint.
		
		Listing 2.95: Minimum Width Constraints
		`.button { min-width: 120px; }`
		`.responsive { min-width: 20%; }`
		`.viewport { min-width: 30vw; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`min-height`	0, px, %, vh	Sets minimum height constraint.
		
		Listing 2.96: Minimum Height Constraints
		`.section { min-height: 100vh; }`
		`.card { min-height: 200px; }`
		`.flexible { min-height: 50%; }`

`margin`	auto, px, %, em	Sets outer spacing around element.
		
		Listing 2.97: Margin Property Examples
		`.center { margin: 0 auto; }`
		`.spaced { margin: 20px; }`
		`.custom { margin: 10px 20px 15px 5px; }`
		`.vertical { margin: 20px 0; }`

`padding`	px, %, em	Sets inner spacing inside element.
		
		Listing 2.98: Padding Property Examples
		`.button { padding: 12px 24px; }`
		`.card { padding: 20px; }`
		`.custom { padding: 10px 20px 15px 5px; }`
		`.percentage { padding: 5%; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`border`	width style color	Sets border shorthand.
		
		Listing 2.99: Border Shorthand Examples
		`.box { border: 1px solid black; }`
		`.thick { border: 3px dashed red; }`
		`.dotted { border: 2px dotted blue; }`
		`.none { border: none; }`

`border-radius`	px, %, em	Creates rounded corners.
		
		Listing 2.100: Border Radius Examples
		`.rounded { border-radius: 8px; }`
		`.circle { border-radius: 50%; }`
		`.pill { border-radius: 25px; }`
		`.custom { border-radius: 10px 20px 5px 15px; }`

`box-shadow`	offset blur spread color	Adds shadow to element.
		
		Listing 2.101: Box Shadow Examples
		`.shadow { box-shadow: 2px 2px 4px rgba(0,0,0,0.3); }`
		`.inset { box-shadow: inset 0 0 10px gray; }`
		`.multiple { box-shadow: 0 2px 4px rgba(0,0,0,0.1), 0 4px 8px rgba(0,0,0,0.2); }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`display`	block, inline, flex, grid, none	Sets display type.
		
		Listing 2.102: Display Property Examples
		`.block { display: block; }`
		`.inline { display: inline; }`
		`.flex { display: flex; }`
		`.grid { display: grid; }`
		`.hidden { display: none; }`
		`.inline-block { display: inline-block; }`

`position`	static, relative, absolute, fixed, sticky	Sets positioning method.
		
		Listing 2.103: Positioning Examples
		`.relative { position: relative; }`
		`.absolute { position: absolute; top: 10px; left: 20px; }`
		`.fixed { position: fixed; bottom: 0; right: 0; }`
		`.sticky { position: sticky; top: 0; }`

`z-index`	auto, integer	Sets stacking order.
		
		Listing 2.104: Z-Index Stacking Examples
		`.modal { z-index: 1000; }`
		`.overlay { z-index: 999; }`
		`.background { z-index: -1; }`
		`.top { z-index: 9999; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`float`	left, right, none	Floats element left or right.
		
		Listing 2.105: Float Property Examples
		`.left { float: left; }`
		`.right { float: right; }`
		`.clear { float: none; clear: both; }`

`overflow`	visible, hidden, scroll, auto	Controls content overflow.
		
		Listing 2.106: Overflow Control Examples
		`.hidden { overflow: hidden; }`
		`.scroll { overflow: scroll; }`
		`.auto { overflow: auto; }`
		`.visible { overflow: visible; }`

`opacity`	0 to 1	Sets transparency level.
		
		Listing 2.107: Opacity Examples
		`.transparent { opacity: 0; }`
		`.semi-transparent { opacity: 0.5; }`
		`.faded { opacity: 0.7; }`
		`.opaque { opacity: 1; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`cursor`	pointer, default, text, crosshair, etc.	Sets cursor appearance.
		
		Listing 2.108: Cursor Style Examples
		`.clickable { cursor: pointer; }`
		`.text { cursor: text; }`
		`.crosshair { cursor: crosshair; }`
		`.not-allowed { cursor: not-allowed; }`

`flex-direction`	row, column, row-reverse, column-reverse	Sets flex container direction.
		
		Listing 2.109: Flexbox Direction Examples
		`.row { display: flex; flex-direction: row; }`
		`.column { display: flex; flex-direction: column; }`
		`.reverse-row { display: flex; flex-direction: row-reverse; }`
		`.reverse-column { display: flex; flex-direction: column-reverse; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`justify-content`	flex-start, center, space-between, etc.	Aligns flex items horizontally.
		
		Listing 2.110: Flexbox Justify Content Examples
		`.start { display: flex; justify-content: flex-start; }`
		`.center { display: flex; justify-content: center; }`
		`.between { display: flex; justify-content: space-between; }`
		`.around { display: flex; justify-content: space-around; }`

`align-items`	stretch, center, flex-start, flex-end	Aligns flex items vertically.
		
		Listing 2.111: Flexbox Align Items Examples
		`.stretch { display: flex; align-items: stretch; }`
		`.center { display: flex; align-items: center; }`
		`.start { display: flex; align-items: flex-start; }`
		`.end { display: flex; align-items: flex-end; }`

`flex-wrap`	nowrap, wrap, wrap-reverse	Controls flex item wrapping.
		
		Listing 2.112: Flexbox Wrap Examples
		`.nowrap { display: flex; flex-wrap: nowrap; }`
		`.wrap { display: flex; flex-wrap: wrap; }`
		`.reverse { display: flex; flex-wrap: wrap-reverse; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example

`flex`	grow shrink basis	Shorthand for flex item properties.
		
		Listing 2.113: Flex Item Properties
		`.flex-1 { flex: 1; }`
		`.flex-2 { flex: 2 0 0; }`
		`.no-grow { flex: 0 1 auto; }`
		`.fixed { flex: 0 0 200px; }`

`grid-template-columns`	track sizes	Defines grid column tracks.
		
		Listing 2.114: CSS Grid Column Templates
		`.grid { display: grid; grid-template-columns: 1fr 1fr 1fr; }`
		`.auto-fit { display: grid; grid-template-columns: repeat(auto-fit, minmax(200px, 1fr)); }`

`grid-template-rows`	track sizes	Defines grid row tracks.
		
		Listing 2.115: CSS Grid Row Templates
		`.grid { display: grid; grid-template-rows: auto 1fr auto; }`
		`.equal-rows { display: grid; grid-template-rows: repeat(3, 1fr); }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example
`grid-gap`	px, em, %	Sets spacing between grid items. Listing 2.116: CSS Grid Gap Examples `.grid { display: grid; grid-gap: 20px; } .custom-gap { display: grid; grid-gap: 10px 20px; }`
`transform`	translate, rotate, scale, skew	Transforms element appearance. Listing 2.117: CSS Transform Examples `.translate { transform: translate(50px, 100px); } .rotate { transform: rotate(45deg); } .scale { transform: scale(1.5); } .combined { transform: translate(10px, 20px) rotate(45deg); }`
`transition`	property duration timing delay	Animates property changes. Listing 2.118: CSS Transition Examples `.smooth { transition: all 0.3s ease; } .color-fade { transition: color 0.5s ease-in-out; } .delayed { transition: opacity 0.3s ease 0.1s; } .multiple { transition: color 0.3s, transform 0.5s; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example
`animation`	name duration timing delay	Defines keyframe animations. Listing 2.119: CSS Animation Examples `@keyframes slideIn { from { transform: translateX(-100%); } to { transform: translateX (0); } } .animated { animation: slideIn 0.5s ease-out; } .infinite { animation: rotate 2s linear infinite; }`
`box-sizing`	content-box, border-box	Controls box model calculation. Listing 2.120: Box Sizing Examples `* { box-sizing: border-box; } .content-box { box-sizing: content-box; } .border-box { box-sizing: border-box; }`
`white-space`	normal, nowrap, pre, pre-wrap	Controls whitespace handling. Listing 2.121: White Space Handling `.nowrap { white-space: nowrap ; } .pre { white-space: pre; } .pre-wrap { white-space: pre- wrap; } .break-spaces { white-space: break-spaces; }`

Continued on next page

Table 2.2 continued from previous page

Property	Common Values	Description and Example
`text-overflow`	clip, ellipsis	Controls text overflow behavior.

Listing 2.122: Text Overflow Examples

```
.ellipsis {
  text-overflow: ellipsis;
  overflow: hidden;
  white-space: nowrap;
}
.clip { text-overflow: clip;
   }
```

2.7. Where JavaScript Fits

JavaScript is the language that adds *behavior* to web pages. HTML provides the **structure** (headings, paragraphs, images), CSS defines the **presentation** (colors, layout, fonts), and JavaScript makes the page **interactive and dynamic** (events, animations, data updates).

2.7.1 Adding JavaScript to HTML

There are three common ways to include JavaScript in an HTML page. Each has different use-cases, advantages, and disadvantages.

2.7.1.1 *1. Inline JavaScript (Discouraged)*

Inline JavaScript is written directly inside an element's attribute, such as `onclick`.

Listing 2.123: Inline JavaScript Example

```
<button onclick="alert('Hello from inline JS!')">Click Me</button>
```

Why discouraged:

- Mixes content (HTML) with behavior (JS) — breaks the principle of separation of concerns.
- Hard to maintain in large projects (logic scattered in multiple attributes).
- Limited functionality: cannot easily reuse or manage complex logic.
- Security issues: inline JS may be blocked by modern Content Security Policies (CSP).

When used: Quick tests, demos, or environments where external files cannot be loaded (rare in modern practice).

2.7.1.2 *2. Internal JavaScript*

Internal JavaScript is written inside a `<script>` block within the same HTML file. It usually goes inside the `<head>` or at the end of `<body>`.

Listing 2.124: Internal JavaScript Example

```
<!DOCTYPE html>
<html>
<head>
  <script>
    document.addEventListener('DOMContentLoaded', () => {
      document.getElementById('btn').addEventListener('click', () => {
        alert('Hello!');
      });
    });
  </script>
</head>
<body>
  <button id="btn">Click Me</button>
</body>
</html>
```

Advantages:

- Keeps JavaScript together in one block (instead of spread in attributes).
- Easy for small projects or single-page experiments.
- Good for teaching examples and quick prototypes.

Disadvantages:

- Logic is still mixed into the HTML file.
- Difficult to reuse across multiple pages.
- Large scripts make HTML file cluttered.

2.7.1.3 *3. External JavaScript (Best Practice)*

In external JavaScript, code is placed in a separate `.js` file and linked with `<script src="...">`. This is the **recommended method** for professional projects.

Listing 2.125: HTML loading external JS

```
<!DOCTYPE html>
<html>
<head>
  <script src="app.js" defer></script>
</head>
<body>
  <button id="btn">Click Me</button>
</body>
</html>
```

Listing 2.126: app.js file

```
// app.js
document.addEventListener('DOMContentLoaded', () => {
  document.getElementById('btn').addEventListener('click', () => {
    alert('Hello from external JS!');
  });
});
```

Advantages:

- Best separation of concerns — HTML for content, CSS for styling, JS for behavior.
- Code reuse: one JS file can serve many HTML pages.
- Easier to maintain and scale.
- Improves performance: browsers cache external JS files.

Disadvantages:

- Requires additional HTTP requests (though minimized by caching/CDN).
- Not ideal for extremely small scripts or quick experiments.

2.7.2 Comparison of Approaches

Method	Where Code Lives	Advantages	Disadvantages
Inline JS	Inside element attribute (`onclick`, `onchange`)	Quick, easy to demo	Messy, insecure, not reusable
Internal JS	`<script>` block in same file	Centralized, good for single page	Not reusable, clutters HTML
External JS	Separate `.js` file linked with `<script src>`	Clean, reusable, scalable, cached by browser	Needs extra file(s), setup

2.7.3 Example: Interactive Demo (Structure)

Below is a compact, best-practice demo showing:

- External CSS and JS files for maintainability.
- DOMContentLoaded to ensure script runs after the page loads.
- Event listeners instead of inline handlers.

Listing 2.127: index.html — compact structure

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,initial-scale=1" />
  <link rel="stylesheet" href="styles.css">
  <title>Interactive Demo</title>
</head>
```

```
<body>
  <div class="container">
    <h1 id="page-title">Interactive Web Page</h1>
    <input id="item-input" placeholder="Enter item...">
    <button id="add-item">Add</button>
    <ul id="item-list"></ul>
  </div>

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

Listing 2.128: app.js — compact behavior

```
// app.js
document.addEventListener('DOMContentLoaded', () => {
  const input = document.getElementById('item-input');
  const addBtn = document.getElementById('add-item');
  const list = document.getElementById('item-list');

  addBtn.addEventListener('click', () => {
    const v = input.value.trim();
    if (!v) return;
    const li = document.createElement('li');
    li.textContent = v;
    list.appendChild(li);
    input.value = '';
  });
});
```

How it works:

1. The HTML provides structure (input, button, list).
2. CSS (not shown here) provides styling.
3. JS waits until the DOM is ready, then binds a click handler.
4. On click, a new list item is created and added dynamically.

2.7.4 Key Takeaways

- **HTML** = structure and semantics.
- **CSS** = presentation and design.
- **JavaScript** = behavior and interactivity.
- Always prefer **external JS files** for maintainable projects.

- Use **event listeners** instead of inline handlers for modern, secure code.

2.8. Bootstrap Introduction and Common CDN Usage Example

Bootstrap is one of the most popular front-end frameworks used for building responsive, mobile-first websites quickly. It provides pre-designed *CSS*, *JavaScript components*, and a *grid system* to make development faster and more consistent.

Introduction to Bootstrap

- Originally developed by Twitter engineers in 2011.
- Helps design responsive layouts using a **12-column grid system**.
- Includes ready-to-use UI components like buttons, forms, navbars, modals, etc.
- Provides utility classes for spacing, typography, colors, alignment, and more.
- Latest version is **Bootstrap 5.3**, which supports dark mode and extended utilities.

Using Bootstrap via CDN

The simplest way to include Bootstrap is through a **Content Delivery Network (CDN)**. This means you don't need to download Bootstrap files locally.

Listing 2.129: Including Bootstrap 5.3 using CDN

```
<!-- Bootstrap CSS -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/
   bootstrap.min.css" rel="stylesheet">

<!-- Bootstrap JS Bundle (with Popper) -->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/
   bootstrap.bundle.min.js"></script>
```

Example Page with Bootstrap

Here is a minimal example showing how to use Bootstrap:

Listing 2.130: Bootstrap CDN Example Page

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0"
     >
  <title>Bootstrap Example</title>
  <!-- Bootstrap CSS -->
```

```
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/
     bootstrap.min.css" rel="stylesheet">
</head>
<body>

  <div class="container text-center my-5">
    <h1 class="display-4">Welcome to Bootstrap 5</h1>
    <p class="lead">This page uses Bootstrap via CDN</p>
    <button class="btn btn-primary btn-lg">Click Me</button>
  </div>

  <!-- Bootstrap JS -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/
     bootstrap.bundle.min.js"></script>
</body>
</html>
```

2.8.1 Bootstrap 5.3 Classes Reference

Table 2.3: Bootstrap 5.3 Classes Reference - Common values and usage examples

Class Type	Common Classes	Description and Example

Container	container, container-fluid, container-sm/md/lg/xl/xxl	Responsive containers for layout structure.
		
		Listing 2.131: Bootstrap Container Examples
		`<div class="container">Fixed width responsive container</div>`
		`<div class="container-fluid">Full width container</div>`
		`<div class="container-md">Container starting at md breakpoint</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
`Grid System`	row, col, col-1 to col-12, col-sm/md/lg/xl/xxl-*	Bootstrap's 12-column grid system. Listing 2.132: Bootstrap Grid System Examples `<div class="row">   <div class="col-12 col-md-6 col-lg-4">Responsive column</div>   <div class="col-6">Half width column</div>   <div class="col">Auto width column</div> </div>`
`Grid Gutters`	g-0 to g-5, gx-*, gy-*	Controls spacing between grid columns. Listing 2.133: Grid Gutter Examples `<div class="row g-3">Gaps between all columns</div> <div class="row gx-4 gy-2">Horizontal and vertical gutters</div> <div class="row g-0">No gutters</div>`
`Display`	d-none, d-block, d-inline, d-flex, d-grid	Controls element display types. Listing 2.134: Display Utility Examples `<div class="d-none d-md-block">Hidden on mobile, block on md+</div> <div class="d-flex">Flexbox container</div> <div class="d-inline-block">Inline block element</div> <div class="d-grid">CSS Grid container</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

`Flexbox Direction`	flex-row, flex-column, flex-row-reverse, flex-column-reverse	Sets flex container direction.
		
		Listing 2.135: Flexbox Direction Examples
		`<div class="d-flex flex-row">Horizontal layout</div>`
		`<div class="d-flex flex-column">Vertical layout</div>`
		`<div class="d-flex flex-row-reverse">Reverse horizontal</div>`
		`<div class="d-flex flex-column-reverse">Reverse vertical</div>`

`Flexbox Justify`	justify-content-start/center/end/between/around/evenly	Controls horizontal alignment in flex container
		
		Listing 2.136: Flexbox Justify Content Examples
		`<div class="d-flex justify-content-center">Center items</div>`
		`<div class="d-flex justify-content-between">Space between</div>`
		`<div class="d-flex justify-content-around">Space around</div>`
		`<div class="d-flex justify-content-evenly">Even spacing</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

`Flexbox Align`	align-items-start/center/end/stretch, align-content-*	Controls vertical alignment in flex containers.
		Listing 2.137: Flexbox Align Items Examples
		`<div class="d-flex align-items-center">Center vertically</div>`
		`<div class="d-flex align-items-start">Align to top</div>`
		`<div class="d-flex align-items-end">Align to bottom</div>`
		`<div class="d-flex align-items-stretch">Stretch to fill</div>`

`Flex Item`	flex-fill, flex-grow-0/1, flex-shrink-0/1, order-0 to order-5	Controls individual flex item behavior.
		Listing 2.138: Flex Item Control Examples
		`<div class="d-flex">`
		`<div class="flex-fill">Fill available space</div>`
		`<div class="flex-grow-1">Grow if space available</div>`
		`<div class="order-2 order-md-1">Change order responsively</div>`
		`</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Margin	m-0 to m-5, mt/mb/ms/me/mx/my-*, m-auto	Sets margins around elements.
		Listing 2.139: Margin Utility Examples
		`<div class="m-3">Margin on all sides</div>`
		`<div class="mt-4 mb-2">Top and bottom margins</div>`
		`<div class="mx-auto">Center horizontally</div>`
		`<div class="ms-lg-5">Left margin on lg+ screens</div>`

Padding	p-0 to p-5, pt/pb/ps/pe/px/py-*	Sets padding inside elements.
		Listing 2.140: Padding Utility Examples
		`<div class="p-4">Padding on all sides</div>`
		`<div class="px-3 py-2">Horizontal and vertical padding</div>`
		`<div class="pt-5">Top padding only</div>`
		`<div class="p-md-4">Padding on medium+ screens</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

`Width`	w-25, w-50, w-75, w-100, w-auto, mw-100	Controls element width.
		Listing 2.141: Width Utility Examples
		`<div class="w-50">50% width</div>`
		`<div class="w-100">Full width</div>`
		`<img src="image.jpg" class="mw-100" alt="Max width image">`
		`<div class="w-auto">Auto width</div>`

`Height`	h-25, h-50, h-75, h-100, vh-100, min-vh-100	Controls element height.
		Listing 2.142: Height Utility Examples
		`<div class="h-100">Full height of parent</div>`
		`<div class="vh-100">Full viewport height</div>`
		`<div class="min-vh-100">Minimum viewport height</div>`
		`<div class="h-50">50% height</div>`

`Text Alignment`	text-start, text-center, text-end, text-sm/md/lg/xl/xxl-*	Aligns text content.
		Listing 2.143: Text Alignment Examples
		`<p class="text-center">Centered text</p>`
		`<p class="text-end">Right aligned text</p>`
		`<p class="text-start text-md-center">Left on mobile, center on md+</p>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Font Weight	fw-bold, fw-normal, fw-light, fw-lighter, fw-bolder	Controls font weight.
		Listing 2.144: Font Weight Examples
		`<p class="fw-bold">Bold text</p>`
		`<p class="fw-light">Light weight text</p>`
		`<p class="fw-normal">Normal weight</p>`
		`<span class="fw-bolder">Extra bold text</span>`

Font Style	fst-italic, fst-normal	Controls font style.
		Listing 2.145: Font Style Examples
		`<p class="fst-italic">Italic text</p>`
		`<em class="fst-normal">Remove italic from em</em>`

Text Transform	text-lowercase, text-uppercase, text-capitalize	Transforms text case.
		Listing 2.146: Text Transform Examples
		`<p class="text-uppercase">UPPERCASE TEXT</p>`
		`<p class="text-lowercase">lowercase text</p>`
		`<p class="text-capitalize">Capitalize Each Word</p>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Text Utilities	text-truncate, lead, small, lh-1/base/lg	Additional text styling utilities.
		Listing 2.147: Text Utility Examples
		`<p class="text-truncate">Long text that will be truncated...</p>`
		`<p class="lead">Lead paragraph with emphasis</p>`
		`<small class="text-muted">Small muted text</small>`
		`<p class="lh-lg">Large line height text</p>`

Text Colors	text-primary, text-secondary, text-success, text-danger, text-warning, text-info, text-light, text-dark, text-muted	Bootstrap color classes for text.
		Listing 2.148: Text Color Examples
		`<p class="text-primary">Primary color text</p>`
		`<p class="text-success">Success color text</p>`
		`<p class="text-danger">Danger color text</p>`
		`<p class="text-muted">Muted gray text</p>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Background Colors	bg-primary, bg-secondary, bg-success, bg-danger, bg-warning, bg-info, bg-light, bg-dark	Bootstrap background color classes.
		
		Listing 2.149: Background Color Examples
		`<div class="bg-primary text-white p-3">Primary background</div>`
		`<div class="bg-success text-white p-3">Success background</div>`
		`<div class="bg-light p-3">Light background</div>`
		`<div class="bg-dark text-white p-3">Dark background</div>`

Background Utilities	bg-body, bg-body-secondary, bg-body-tertiary, bg-gradient	Bootstrap 5.3 background utilities.
		
		Listing 2.150: Background Utility Examples
		`<div class="bg-body p-3">Body background color</div>`
		`<div class="bg-body-secondary p-3">Secondary body bg</div>`
		`<div class="bg-primary bg-gradient p-3">Gradient overlay</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

`Text-Background`	text-bg-primary, text-bg-secondary, text-bg-success, etc.	Bootstrap 5.3 combined text and background classes.
		
		Listing 2.151: Text-Background Examples
		`<span class="text-bg-primary p-2">Primary text on primary bg</span>`
		`<span class="text-bg-success p-2">Success text on success bg</span>`
		`<span class="text-bg-warning p-2">Warning text on warning bg</span>`

`Opacity`	opacity-10, opacity-25, opacity-50, opacity-75, opacity-100	Controls element transparency.
		
		Listing 2.152: Opacity Examples
		`<div class="opacity-25">25% opacity</div>`
		`<div class="opacity-50">50% opacity</div>`
		`<div class="opacity-75">75% opacity</div>`
		`<img src="image.jpg" class="opacity-50" alt="Semi-transparent">`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
`Position`	position-static, position-relative, position-absolute, position-fixed, position-sticky	Controls element positioning. Listing 2.153: Position Examples `<div class="position-relative"><div class="position-absolute top-0 end-0">Absolute positioned</div></div><div class="position-fixed bottom-0 end-0">Fixed position</div><div class="position-sticky top-0">Sticky header</div>`
`Position Offsets`	top-0/50/100, bottom-0/50/100, start-0/50/100, end-0/50/100	Controls position offset values. Listing 2.154: Position Offset Examples `<div class="position-absolute top-0 start-0">Top left corner</div><div class="position-absolute bottom-0 end-0">Bottom right corner</div><div class="position-absolute top-50 start-50">Center position</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Z-Index	z-n1, z-0, z-1, z-2, z-3	Controls stacking order (Bootstrap 5.3 expanded).
		Listing 2.155: Z-Index Examples
		`<div class="position-relative z-3">Highest layer</div>`
		`<div class="position-absolute z-2">Middle layer</div>`
		`<div class="position-absolute z-1">Lower layer</div>`
		`<div class="position-relative z-n1">Below normal flow</div>`

Overflow	overflow-auto, overflow-hidden, overflow-scroll, overflow-visible	Controls content overflow.
		Listing 2.156: Overflow Examples
		`<div class="overflow-auto" style="height: 100px;">Scrollable content</div>`
		`<div class="overflow-hidden">Hidden overflow</div>`
		`<div class="overflow-scroll">Always show scrollbars</div>`

Visibility	visible, invisible	Controls element visibility without affecting layout.
		Listing 2.157: Visibility Examples
		`<div class="visible">Visible element</div>`
		`<div class="invisible">Hidden but takes space</div>`
		`<div class="d-none">Completely hidden</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Borders	border, border-0, border-top/bottom/start/end, border-primary/secondary/etc.	Border utilities.
		Listing 2.158: Border Examples
		`<div class="border p-3">All borders</div>`
		`<div class="border-top border-primary p-3">Top border only</div>`
		`<div class="border-0 p-3">Remove all borders</div>`
		`<div class="border-3 border-success p-3">Thick green border</div>`

Border Radius	rounded, rounded-0 to rounded-3, rounded-circle, rounded-pill, rounded-top/bottom/start/end	Rounded corner utilities.
		Listing 2.159: Border Radius Examples
		`<div class="rounded p-3">Rounded corners</div>`
		`<div class="rounded-circle" style="width: 50px; height: 50px;">Circle</div>`
		`<div class="rounded-pill p-2">Pill shape</div>`
		`<div class="rounded-top p-3">Rounded top only</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Shadows	shadow-none, shadow-sm, shadow, shadow-lg	Box shadow utilities.
		Listing 2.160: Shadow Examples
		`<div class="shadow-sm p-3 mb-3">Small shadow</div>`
		`<div class="shadow p-3 mb-3">Regular shadow</div>`
		`<div class="shadow-lg p-3 mb-3">Large shadow</div>`
		`<div class="shadow-none p-3">No shadow</div>`

Buttons	btn, btn-primary/secondary/success/danger/warning/info/light/dark, btn-outline-*, btn-sm/lg	Bootstrap button classes.
		Listing 2.161: Button Examples
		`<button class="btn btn-primary">Primary Button</button>`
		`<button class="btn btn-outline-secondary">Outline Button</button>`
		`<button class="btn btn-success btn-lg">Large Success</button>`
		`<button class="btn btn-danger btn-sm">Small Danger</button>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

`Form Controls`	form-control, form-control-sm/lg, form-select, form-check	Form input styling.
		
		Listing 2.162: Form Control Examples
		`<input type="text" class="form-control" placeholder="Text input">`
		`<select class="form-select">`
		`<option>Choose option</option>`
		`</select>`
		`<div class="form-check">`
		`<input class="form-check-input" type="checkbox" id="check1">`
		`<label class="form-check-label" for="check1">Check me</label>`
		`</div>`

`Form Validation`	is-valid, is-invalid, valid-feedback, invalid-feedback	Form validation styling.
		
		Listing 2.163: Form Validation Examples
		`<input type="email" class="form-control is-valid" value="test@example.com">`
		`<div class="valid-feedback">Looks good!</div>`
		
		`<input type="email" class="form-control is-invalid" value="invalid-email">`
		`<div class="invalid-feedback">Please provide a valid email.</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
Floating Labels	form-floating	Bootstrap floating label forms. Listing 2.164: Floating Label Examples `<div class="form-floating mb-3"><input type="email" class="form-control" id="floatingInput" placeholder="name@example.com"><label for="floatingInput">Email address</label></div>`
Input Groups	input-group, input-group-text	Group inputs with text or buttons. Listing 2.165: Input Group Examples `<div class="input-group"><span class="input-group-text">@</span><input type="text" class="form-control" placeholder="Username"></div><div class="input-group"><input type="text" class="form-control"><button class="btn btn-primary" type="button">Button</button></div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
Cards	card, card-body, card-header, card-footer, card-title, card-text	Bootstrap card components. Listing 2.166: Card Examples `<div class="card">   <div class="card-header">   Card Header</div>   <div class="card-body">   <h5 class="card-title">   Card Title</h5>   <p class="card-text">Card content goes here.</p>   </div>   <div class="card-footer">   Card Footer</div> </div>`
Alerts	alert, alert-primary/secondary/success/danger/warning/info/light/dark, alert-dismissible	Bootstrap alert messages. Listing 2.167: Alert Examples `<div class="alert alert-success">Success message!</div> <div class="alert alert-danger">Error message!</div> <div class="alert alert-warning alert-dismissible fade show">   Warning message with close button   <button type="button" class="btn-close" data-bs-dismiss="alert"></button> </div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Badges	badge, bg-*, rounded-pill	Bootstrap badge components.
		Listing 2.168: Badge Examples
		`<span class="badge bg-primary">Primary Badge</span>`
		`<span class="badge bg-success rounded-pill">Pill Badge</span>`
		`<h1>Example heading <span class="badge bg-secondary">New</span></h1>`

List Groups	list-group, list-group-item, list-group-item-action, list-group-flush	Bootstrap list group components.
		Listing 2.169: List Group Examples
		`<ul class="list-group">`
		`<li class="list-group-item">Regular item</li>`
		`<li class="list-group-item active">Active item</li>`
		`<li class="list-group-item list-group-item-action">Actionable item</li>`
		`</ul>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
`Navigation`	nav, nav-tabs, nav-pills, nav-justified, nav-item, nav-link	Bootstrap navigation components. Listing 2.170: Navigation Examples `<ul class="nav nav-tabs"><li class="nav-item"><a class="nav-link active" href="#">Active Tab</a></li><li class="nav-item"><a class="nav-link" href="#">Tab 2</a></li></ul>`
`Navbar`	navbar, navbar-expand-*, navbar-brand, navbar-nav, navbar-toggler	Bootstrap navbar components. Listing 2.171: Navbar Examples `<nav class="navbar navbar-expand-lg navbar-dark bg-dark"><div class="container"><a class="navbar-brand" href="#">Brand</a><button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav"><span class="navbar-toggler-icon"></span></button></div></nav>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
`Modal`	modal, modal-dialog, modal-content, modal-header, modal-body, modal-footer	Bootstrap modal components.

Listing 2.172: Modal Examples

```
<div class="modal fade" id="
    exampleModal">
  <div class="modal-dialog">
    <div class="modal-content
        ">
      <div class="modal-
          header">
        <h5 class="modal-
            title">Modal Title
            </h5>
        <button type="button"
            class="btn-close"
            data-bs-dismiss="
            modal"></button>
      </div>
      <div class="modal-body"
          >Modal content goes
          here.</div>
    </div>
  </div>
</div>
```

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
`Dropdown`	dropdown, dropdown-menu, dropdown-item, dropdown-toggle	Bootstrap dropdown components.

Listing 2.173: Dropdown Examples

```
<div class="dropdown">
  <button class="btn btn-
     primary dropdown-toggle"
      type="button"
          data-bs-toggle="
             dropdown">
    Dropdown button
  </button>
  <ul class="dropdown-menu">
    <li><a class="dropdown-
       item" href="#">Action<
       /a></li>
    <li><a class="dropdown-
       item" href="#">Another
        action</a></li>
  </ul>
</div>
```

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
`Accordion`	accordion, accordion-item, accordion-header, accordion-button, accordion-collapse, accordion-body	Bootstrap accordion (collapsible) components.

Listing 2.174: Accordion Examples

```
<div class="accordion" id="
   accordionExample">
  <div class="accordion-item"
     >
    <h2 class="accordion-
       header">
      <button class="
         accordion-button"
         type="button"
              data-bs-toggle=
                 "collapse"
                 data-bs-
                 target="#
                 collapseOne"
                 >
        Accordion Item #1
      </button>
    </h2>
    <div id="collapseOne"
       class="accordion-
       collapse collapse show
       ">
      <div class="accordion-
         body">Content goes
         here.</div>
    </div>
  </div>
</div>
```

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
Progress	progress, progress-bar, progress-bar-striped, progress-bar-animated	Bootstrap progress bars. Listing 2.175: Progress Bar Examples `<div class="progress">   <div class="progress-bar" style="width: 50%"></div> </div> <div class="progress">   <div class="progress-bar progress-bar-striped progress-bar-animated bg-success" style="width: 75%"></div> </div>`
Spinners	spinner-border, spinner-grow, spinner-border-sm, spinner-grow-sm	Bootstrap loading spinners. Listing 2.176: Spinner Examples `<div class="spinner-border" role="status">   <span class="visually-hidden">Loading...</span> </div> <div class="spinner-grow text-primary" role="status">   <span class="visually-hidden">Loading...</span> </div> <div class="spinner-border spinner-border-sm" role="status"></div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Tables	table, table-striped, table-bordered, table-hover, table-sm, table-responsive	Bootstrap table styling.
		Listing 2.177: Table Examples
		`<table class="table table-striped table-hover">`
		`<thead class="table-dark">`
		`<tr>`
		`<th>Name</th>`
		`<th>Email</th>`
		`</tr>`
		`</thead>`
		`<tbody>`
		`<tr>`
		`<td>John Doe</td>`
		`<td>john@example.com</td>`
		`</tr>`
		`</tbody>`
		`</table>`

Utilities	text-decoration-none, user-select-none, pe-none, cursor-pointer	Additional utility classes.
		Listing 2.178: Utility Examples
		`<a href="#" class="text-decoration-none">Link without underline</a>`
		`<div class="user-select-none">Non-selectable text</div>`
		`<div class="pe-none">Pointer events disabled</div>`
		`<div class="cursor-pointer">Pointer cursor</div>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example

Image Utilities	img-fluid, img-thumbnail, figure, figure-img, figure-caption	Image styling utilities.
		
		Listing 2.179: Image Utility Examples
		`<img src="image.jpg" class="img-fluid" alt="Responsive image">`
		`<img src="image.jpg" class="img-thumbnail" alt="Thumbnail">`
		`<figure class="figure">`
		`<img src="image.jpg" class="figure-img img-fluid" alt="Figure">`
		`<figcaption class="figure-caption">Image caption</figcaption>`
		`</figure>`

Dark Mode	data-bs-theme="dark/light"	Bootstrap 5.3 color mode support.
		
		Listing 2.180: Dark Mode Examples
		`<html data-bs-theme="dark">`
		`<body data-bs-theme="light">`
		`<div class="bg-body text-body p-3">Adapts to theme</div>`
		`<div class="card" data-bs-theme="dark">Dark themed card</div>`
		`</body>`
		`</html>`

Continued on next page

Table 2.3 continued from previous page

Class Type	Common Classes	Description and Example
`Responsive Patterns`	Breakpoint-specific classes	Examples of responsive class patterns. Listing 2.181: Responsive Pattern Examples `<!-- Hide on mobile, show on tablets+ --> <div class="d-none d-md-block">Desktop content</div> <!-- Responsive columns --> <div class="col-12 col-sm-6 col-lg-4">Responsive column</div> <!-- Responsive text alignment --> <p class="text-center text-md-start">Center on mobile, left on desktop</p> <!-- Responsive spacing --> <div class="m-2 m-lg-4">Larger margins on large screens</div>`

Chapter Summary

In this chapter, we learned:

- HTML provides semantic structure.
- CSS controls visual presentation and layout.
- JavaScript adds interactivity and dynamic behavior.
- Inline JavaScript is discouraged; internal is good for small projects; external is best practice.
- Use DOMContentLoaded and `addEventListener()` to ensure scripts run safely.

Next Chapter Preview: We will explore JavaScript fundamentals — variables, data types, operators, functions, and control structures — with hands-on examples and exercises.

CHAPTER 3

3 JAVASCRIPT FUNDAMENTALS

3.1. Basic Instructions and Statements

3.1.1 Understanding JavaScript Statements

In JavaScript, a **statement** is a complete instruction that performs an action. Think of statements as sentences in the JavaScript language - they tell the computer what to do step by step.

Key Characteristics of Statements:

- Each statement performs a specific action
- Statements are executed in order from top to bottom
- Most statements end with a semicolon (;) - though JavaScript can often insert them automatically
- Statements can be grouped together using curly braces {}

3.1.2 Types of JavaScript Statements

JavaScript has several types of statements, each serving different purposes:

Table 3.1: Types of JavaScript Statements

Statement Type	Purpose	Example
Expression Statement	Evaluates an expression	`x = 5; console.log("Hello");`
Declaration Statement	Declares variables or functions	`let name; function greet() {}`
Control Flow Statement	Controls program execution	`if, for, while, switch`
Jump Statement	Changes execution flow	`break, continue, return`
Block Statement	Groups multiple statements	`{ statement1; statement2; }`

3.1.3 Statement Examples

Listing 3.1: Basic JavaScript Statements

```
// Expression statements
let age = 25;                    // Assignment statement
console.log("Welcome!");         // Function call statement
```

```
age + 5;                              // Expression statement (evaluates but
    doesn't store)

// Declaration statements
let userName;                         // Variable declaration
function sayHello() {                 // Function declaration
    return "Hello World!";
}

// Block statement
{
    let localVar = "I'm in a block";
    console.log(localVar);
}

// Control flow statements (covered in detail later)
if (age >= 18) {
    console.log("You are an adult");
}
```

3.2. Comments

3.2.1 Purpose of Comments

Comments are text in your code that JavaScript ignores during execution. They serve several important purposes:

- **Documentation:** Explain what your code does
- **Communication:** Help other developers understand your logic
- **Debugging:** Temporarily disable code without deleting it
- **Planning:** Outline your approach before writing code
- **Maintenance:** Provide context for future modifications

3.2.2 Types of Comments

JavaScript supports two types of comments:

Table 3.2: JavaScript Comment Types

Comment Type	Syntax	Use Case
Single-line	`// Comment text`	Brief explanations, inline notes
Multi-line	`/* Comment text */`	Longer explanations, function documentation
JSDoc	`/** @param {type} name */`	Professional documentation

3.2.3 Comment Examples and Best Practices

Listing 3.2: JavaScript Comments Examples

```
// Single-line comment explaining the next line
let pi = 3.14159;  // Mathematical constant

/*
 * Multi-line comment for complex explanations
 * This function calculates the area of a circle
 * using the formula: $A = \pi r^{2}$.
 */
function calculateCircleArea(radius) {
    return pi * radius * radius;
}

/**
 * JSDoc comment for professional documentation
 * @param {number} length - The length of the rectangle
 * @param {number} width - The width of the rectangle
 * @returns {number} The area of the rectangle
 */
function calculateRectangleArea(length, width) {
    // Input validation
    if (length <= 0 || width <= 0) {
        return 0;  // Return 0 for invalid dimensions
    }

    // Calculate and return area
    return length * width;
}

// TODO: Add error handling for negative numbers
// FIXME: This function doesn't handle decimal places properly
// NOTE: Consider using a math library for precision
```

3.3. Variables

3.3.1 What are Variables?

Variables are containers that store data values. Think of them as labeled boxes where you can put

information and retrieve it later. In JavaScript, variables can hold different types of data and their values can change during program execution.

3.3.2 Variable Declaration Keywords

JavaScript provides three keywords for declaring variables, each with different characteristics:

Table 3.3: JavaScript Variable Declaration Keywords

Keyword	Scope	Reassignment	Use Case
`var`	Function or global	Yes	Legacy code (avoid in modern JS)
`let`	Block scope	Yes	When value needs to change
`const`	Block scope	No	For constants and references

3.3.3 Variable Declaration and Assignment

Listing 3.3: Variable Declaration Examples

```
// Variable declaration without initialization
let userName;                // undefined until assigned
let age;                     // undefined until assigned

// Variable declaration with initialization
let firstName = "John";      // String value
let lastName = "Doe";        // String value
let currentAge = 25;         // Number value
let isStudent = true;        // Boolean value

// Constant declaration (must be initialized)
const PI = 3.14159;          // Mathematical constant
const MAX_USERS = 100;       // Configuration constant

// Multiple variable declarations
let x = 5, y = 10, z = 15;   // Declare multiple variables

// Variable reassignment
age = 26;                    // Change the value
firstName = "Jane";          // Change the value

// This would cause an error:
// PI = 3.14;                // TypeError: Assignment to constant
   variable
```

3.3.4 Variable Naming Rules and Conventions

Understanding proper variable naming is crucial for writing maintainable code:

Table 3.4: Variable Naming Rules and Conventions

Category	Rule/Convention	Example
Must start with	Letter, underscore, or $	`name, _id, $element`
Cannot start with	Number	`1name` X
Can contain	Letters, numbers, _, $	`user_1, element$`
Cannot contain	Spaces or special chars	`user name` X
Case sensitivity	Matters	`Name != name`
Reserved words	Cannot use keywords	`let, if, function` X
Convention	camelCase for variables	`firstName, totalAmount`
Convention	UPPER_CASE for constants	`MAX_SIZE, API_URL`

Listing 3.4: Variable Naming Examples

```
// Good variable names
let firstName = "John";          // Clear and descriptive
let userAge = 25;                // Follows camelCase
let isLoggedIn = false;          // Boolean with "is" prefix
let shoppingCart = [];           // Array for storing items
let API_BASE_URL = "https://api.example.com";  // Constant

// Poor variable names (avoid these)
let a = "John";                  // Too short, not descriptive
let user_name = "John";          // snake_case (not JS convention)
let UserAge = 25;                // PascalCase (use for classes)
let x = true;                    // Meaningless name
let temp = [];                   // Vague purpose

// Reserved words (cannot use as variable names)
// let if = 5;                   // Error: Unexpected token
// let function = "test";        // Error: Unexpected token
// let var = 10;                 // Error: Unexpected token
```

3.4. Data Types

3.4.1 Understanding JavaScript Data Types

JavaScript has dynamic typing, meaning variables can hold different types of values and change types during execution. JavaScript data types are divided into two main categories:

- **Primitive Types:** Basic data types that store simple values
- **Non-Primitive Types:** Complex data types that store collections of values or more complex entities

3.4.2 Primitive Data Types

Table 3.5: JavaScript Primitive Data Types

Type	Description	Example Values	Use Cases
`number`	Integer or floating-point	`42, 3.14, -17, Infinity`	Calculations, counters, measurements
`string`	Sequence of characters	`"Hello", 'World', `Template``	Text, messages, identifiers
`boolean`	True or false	`true, false`	Conditions, flags, switches
`undefined`	Declared but not assigned	`undefined`	Default value for variables
`null`	Intentional absence of value	`null`	Resetting variables, empty state
`symbol`	Unique identifier	`Symbol('id')`	Object property keys
`bigint`	Large integers	`123n, BigInt(123)`	Very large numbers

3.4.3 Working with Numbers

Listing 3.5: JavaScript Numbers

```
// Integer numbers
let age = 25;
let temperature = -5;
let score = 0;

// Floating-point numbers
let price = 19.99;
let pi = 3.14159;
let percentage = 0.85;

// Special number values
let infinity = Infinity;          // Positive infinity
let negInfinity = -Infinity;      // Negative infinity
let notANumber = NaN;             // Not a Number

// Number operations
let sum = 10 + 5;                 // Addition: 15
let difference = 10 - 5;          // Subtraction: 5
let product = 10 * 5;             // Multiplication: 50
let quotient = 10 / 5;            // Division: 2
let remainder = 10 % 3;           // Modulus: 1
let power = 2 ** 3;               // Exponentiation: 8

// Number methods and properties
```

```
console.log(Number.MAX_VALUE);          // Largest number
console.log(Number.MIN_VALUE);          // Smallest positive number
console.log(Number.isInteger(42));      // true
console.log(Number.isNaN(NaN));         // true
console.log(parseFloat("3.14abc"));     // 3.14
console.log(parseInt("42px"));          // 42
```

3.4.4 Working with Strings

Listing 3.6: JavaScript Strings

```
// String declaration methods
let singleQuotes = 'Hello World';
let doubleQuotes = "Hello World";
let templateLiteral = `Hello World`;

// String concatenation
let firstName = "John";
let lastName = "Doe";
let fullName = firstName + " " + lastName;  // "John Doe"

// Template literals (modern approach)
let greeting = `Hello, ${firstName} ${lastName}!`;
let multiLine = `This is a
multi-line
string`;

// String properties and methods
let message = "JavaScript Programming";
console.log(message.length);                // 21 (property)
console.log(message.toUpperCase());         // "JAVASCRIPT PROGRAMMING"
console.log(message.toLowerCase());         // "javascript programming"
console.log(message.charAt(0));             // "J"
console.log(message.indexOf("Script"));     // 4
console.log(message.substring(0, 10));      // "JavaScript"
console.log(message.slice(-11));            // "Programming"
console.log(message.replace("Java", "Type")); // "TypeScript
   Programming"

// String escape characters
let escaped = "She said, \"Hello!\"";       // She said, "Hello!"
let newLine = "Line 1\nLine 2";             // Line break
let tab = "Column1\tColumn2";               // Tab character
let backslash = "Path: C:\\Users\\";        // Backslash
```

3.4.5 Working with Booleans

Listing 3.7: JavaScript Booleans

```
// Boolean values
let isActive = true;
let isComplete = false;

// Boolean from expressions
let isAdult = age >= 18;            // true if age is 18 or more
let hasPermission = true && isActive;  // Logical AND
let canAccess = isAdult || hasPermission;  // Logical OR
let isInactive = !isActive;         // Logical NOT

// Truthy and Falsy values
// Falsy values: false, 0, "", null, undefined, NaN
// Everything else is truthy

let truthyValues = [1, "hello", [], {}, "0", "false"];
let falsyValues = [false, 0, "", null, undefined, NaN];

// Boolean conversion
console.log(Boolean(1));            // true
console.log(Boolean(0));            // false
console.log(Boolean("hello"));      // true
console.log(Boolean(""));           // false
console.log(!!42);                  // true (double negation)
console.log(!!"");                  // false

// Comparison operators return booleans
let a = 5, b = 10;
console.log(a == b);                // false (equal value)
console.log(a === b);               // false (equal value and type)
console.log(a != b);                // true (not equal value)
console.log(a !== b);               // true (not equal value or type)
console.log(a < b);                 // true (less than)
console.log(a > b);                 // false (greater than)
console.log(a <= b);                // true (less than or equal)
console.log(a >= b);                // false (greater than or equal)
```

3.4.6 Special Values: null and undefined

Listing 3.8: null and undefined

```
// undefined: variable declared but not assigned
let notAssigned;
console.log(notAssigned);           // undefined
console.log(typeof notAssigned);    // "undefined"

// null: intentional absence of value
let empty = null;
console.log(empty);                 // null
console.log(typeof empty);          // "object" (this is a known quirk)

// Differences between null and undefined
console.log(null == undefined);     // true (loose equality)
console.log(null === undefined);    // false (strict equality)

// Checking for null or undefined
function checkValue(value) {
    if (value == null) {            // Checks for both null and
       undefined
        return "No value";
    }
    if (value === null) {           // Checks specifically for null
        return "Explicitly null";
    }
    if (value === undefined) {      // Checks specifically for undefined
        return "Undefined";
    }
    return "Has value";
}

// Default values for null/undefined
let userName = null;
let displayName = userName || "Guest";  // "Guest"
let safeName = userName ?? "Default";   // "Default" (nullish
   coalescing)
```

3.4.7 Type Checking and Conversion

Listing 3.9: Type Checking and Conversion

```
// typeof operator
console.log(typeof 42);             // "number"
console.log(typeof "hello");        // "string"
console.log(typeof true);           // "boolean"
```

```
console.log(typeof undefined);      // "undefined"
console.log(typeof null);           // "object" (quirk)
console.log(typeof []);             // "object"
console.log(typeof {});             // "object"
console.log(typeof function(){});   // "function"

// Type conversion (implicit)
let result1 = "5" + 3;              // "53" (string concatenation)
let result2 = "5" - 3;              // 2 (numeric subtraction)
let result3 = "5" * 3;              // 15 (numeric multiplication)
let result4 = +"5";                 // 5 (unary plus converts to number)

// Type conversion (explicit)
let str = "123";
let num = Number(str);              // 123
let bool = Boolean(str);            // true

let value = 456;
let stringValue = String(value);    // "456"
let stringValue2 = value.toString(); // "456"

// Parsing strings to numbers
console.log(parseInt("123.45"));    // 123 (integer part)
console.log(parseFloat("123.45")); // 123.45 (full number)
console.log(parseInt("123abc"));    // 123 (stops at first non-digit)
console.log(parseInt("abc123"));    // NaN (starts with non-digit)
```

3.5. Arrays

3.5.1 Introduction to Arrays ›

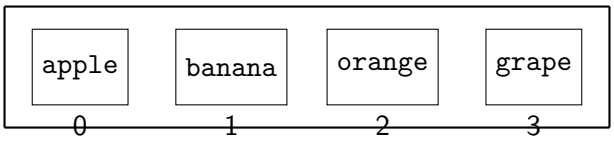
Arrays are ordered collections of values. They allow us to store multiple items in a single variable, instead of creating separate variables for each item.

Key Characteristics:

- **Ordered:** Items are arranged in a fixed sequence.
- **Indexed:** Each item is identified by an index (starting at 0).
- **Dynamic:** Arrays can grow and shrink during program execution.
- **Mixed Types:** An array may contain numbers, strings, objects, or even other arrays.
- **Zero-Based:** The first element is always at index 0.

Visual Representation:

Array: fruits



Each box represents an array element, and the number below is its index. For example: `fruits[2]` gives `"orange"`.

3.5.2 Array Creation and Basic Operations

Arrays can be created using literals (preferred) or the `Array` constructor.

Listing 3.10: Creating Arrays

```
// Array literals
let fruits = ["apple", "banana", "orange"];
let numbers = [10, 20, 30, 40];
let mixed = ["hello", 42, true, null];

// Empty array
let empty = [];

// Using Array constructor
let created = new Array(3);        // [empty x 3]
let fromValues = new Array("x", "y", "z"); // ["x","y","z"]

console.log(fruits[0]);   // "apple"
console.log(numbers[2]);  // 30
```

Listing 3.11: Modifying and Accessing Arrays

```
let fruits = ["apple", "banana", "orange"];

fruits[1] = "grape";  // replace "banana" with "grape"
fruits[3] = "kiwi";   // add at index 3

console.log(fruits);   // ["apple", "grape", "orange", "kiwi"]
console.log(fruits.length); // 4
```

Listing 3.12: Adding and Removing Elements

```
let colors = ["red", "green"];

// Add to end
```

```
colors.push("blue");
console.log(colors);  // ["red", "green", "blue"]

// Remove last element
let last = colors.pop();
console.log(last);    // "blue"

// Add to beginning
colors.unshift("yellow");
console.log(colors);  // ["yellow", "red", "green"]

// Remove first element
let first = colors.shift();
console.log(first);   // "yellow"
```

3.5.3 Array Methods Reference

Table 3.6 lists the most important array methods with explanations and examples.

Table 3.6: Essential JavaScript Array Methods

Method	Purpose	Example
`push()`	Add to end	`arr.push("new");`
`pop()`	Remove last	`let last = arr.pop();`
`unshift()`	Add to start	`arr.unshift("first");`
`shift()`	Remove first	`let first = arr.shift();`
`slice()`	Extract copy	`arr.slice(1,3);`
`splice()`	Insert/remove	`arr.splice(2,1,"new");`
`indexOf()`	Find index	`arr.indexOf("apple");`
`includes()`	Contains?	`arr.includes("apple");`
`join()`	Array → string	`arr.join(", ");`
`reverse()`	Reverse order	`arr.reverse();`
`sort()`	Sort elements	`arr.sort();`
`concat()`	Merge arrays	`arr1.concat(arr2);`
`forEach()`	Loop elements	`arr.forEach(x => ...);`
`map()`	Transform	`arr.map(x => x*2);`
`filter()`	Select subset	`arr.filter(x => x>10);`
`find()`	First match	`arr.find(x => x>10);`
`reduce()`	Single value	`arr.reduce((a,b)=>a+b,0);`

3.5.4 Advanced Array Operations

Listing 3.13: Iteration and Transformation

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

```
// forEach
numbers.forEach((num, i) => console.log(i, num));

// map - create new transformed array
let squares = numbers.map(n => n * n);
console.log(squares);  // [1,4,9,16,25]

// filter - select even numbers
let evens = numbers.filter(n => n % 2 === 0);
console.log(evens);    // [2,4]
```

Listing 3.14: Searching and Reducing

```
let nums = [10, 20, 30, 40];

// find
let found = nums.find(x => x > 15);
console.log(found); // 20

// reduce - sum all elements
let sum = nums.reduce((total, n) => total + n, 0);
console.log(sum); // 100
```

Listing 3.15: Sorting and Multi-Dimensional Arrays

```
let students = [
    { name: "Alice", grade: 85 },
    { name: "Bob", grade: 92 },
    { name: "Charlie", grade: 78 }
];

// sort by grade descending
students.sort((a, b) => b.grade - a.grade);
console.log(students);

// 2D array (matrix)
let matrix = [
    [1,2,3],
    [4,5,6],
    [7,8,9]
];
console.log(matrix[1][2]); // 6
```

3.6. Strings (Advanced)

3.6.1 Overview and Key Concepts

Strings in JavaScript are immutable sequences of UTF-16 code units. That means when you "change" a string you actually create a new one. Modern JavaScript provides rich methods for inspection, transformation, searching, and formatting. Important topics to understand:

- **Immutability:** Methods return new strings; originals remain unchanged.
- **UTF-16 / Unicode:** Characters outside the Basic Multilingual Plane (e.g. emoji) use surrogate pairs. Simple `.length` counts UTF-16 code units, not Unicode code points.
- **Template Literals:** Backtick strings with interpolation, multi-line support and tagged templates.
- **Regular Expressions:** Powerful for pattern matching and replacing (use the `u` flag for Unicode).
- **Normalization:** Use `String.prototype.normalize()` for canonical Unicode comparisons.
- **Performance:** Prefer `Array.join` or streaming when assembling very large strings; avoid repeated concatenation in hot loops.

3.6.2 Character basics, indices and Unicode

Indexing and code points: `str[i]` and `charAt(i)` access UTF-16 code units. Characters like ☺ may occupy two code units.

Listing 3.16: UTF-16 surrogate pairs and counting code points

```
// Example with emoji (surrogate pair)
let s = "A\u{1F600}B";      // "AB"
console.log(s);             // AB
console.log(s.length);      // 3 (UTF-16 code units: 'A', high+low
   surrogate, 'B')

// To count actual Unicode code points:
let codePoints = [...s].length; // spread uses iterator over code
   points
console.log("Code points:", codePoints); // 3

// Accessing the emoji correctly:
console.log([...s][1]); //  ""
```

Note: When working with Unicode-aware processing (emoji, accents), prefer spread `[...str]`, `for ... of`, or use ES2015 methods that understand code points.

3.6.3 String Methods — Expanded examples

Listing 3.17: Common string operations (examples)

```
// Basic transformations
let t = "  Café    ";
console.log(t.trim());                  // "Café  "
console.log(t.toUpperCase());           // "  CAFÉ    "
console.log(t.toLowerCase());           // "  café    "

// slice, substring, substr
console.log(t.slice(2, 6));             // "Caf" (note indices refer to
   UTF-16 code units)
console.log(t.substring(2, 6));         // similar behavior for positive
   indices

// padStart / padEnd
console.log("42".padStart(5, "0"));     // "00042"
console.log("42".padEnd(5, "."));       // "42..."

// repeat
console.log("ha".repeat(3));            // "hahaha"
```

3.6.4 Searching and replacing (with RegExp) ›

Regular expressions let you perform powerful searches and replacements.

Listing 3.18: RegExp: search, replace, global, flags

```
// Simple replace (first occurrence)
let text = "red blue red";
console.log(text.replace("red", "green")); // "green blue red"

// Replace all using regex with global flag
console.log(text.replace(/red/g, "green")); // "green blue green"

// Using capture groups and backreferences
let names = "Doe, John; Smith, Jane";
let swapped = names.replace(/(\w+),\s*(\w+)/g, "$2 $1");
console.log(swapped); // "John Doe; Jane Smith"

// Unicode-aware regex (u flag) and case-insensitive (i)
let emojiText = "  snowman";
console.log(/\p{Script=Latin}/u.test("a")); // true if engine supports
   Unicode property escapes

// Using replace with function (complex replacements)
let prices = "Price: $5, $15, $100";
```

```
let total = 0;
let replaced = prices.replace(/\$(\d+)/g, (m, p1) => {
    total += Number(p1);
    return "€" + (Number(p1) * 0.9).toFixed(2);
});
console.log(replaced); // converted prices in Euros
console.log("Total USD:", total);
```

Note: Use `replaceAll()` for literal all-occurrence replacement (ES2021). For patterns, use regex with g.

3.6.5 Template literals and tagged templates

Template literals simplify multi-line strings and interpolation. Tagged templates allow custom parsing/-formatting.

Listing 3.19: Template literals and tagged templates

```
// Basic template literal
let user = "Alice";
let greeting = `Hello, ${user}! Today is ${new Date().
   toLocaleDateString()}.`;
console.log(greeting);

// Multi-line
let poem = `Roses are red,
Violets are blue,
Template literals
Make strings easy to view.`;
console.log(poem);

// Tagged template (custom sanitizer)
function htmlEscape(strings, ...values) {
  return strings.reduce((acc, s, i) => {
    let val = values[i - 1];
    // Basic escape (for illustration)
    if (val !== undefined) {
      val = String(val)
        .replaceAll("&", "&amp;")
        .replaceAll("<", "&lt;")
        .replaceAll(">", "&gt;")
        .replaceAll('"', "&quot;");
    }
    return acc + (val !== undefined ? val : "") + s;
  });
}
```

```
let unsafe = "<script>alert('x')</script>";
let safe = htmlEscape`User input: ${unsafe}`;
console.log(safe); // "User input: &lt;script&gt;alert('x')&lt;/script&
    gt;"
```

3.6.6 Normalization and Unicode comparisons

Different Unicode sequences can represent the same human-readable character (e.g., "é" as single code point or "e" + combining accent). Use `normalize()` before comparing.

Listing 3.20: Normalization example

```
let a = "é";                    // single code point U+00E9
let b = "e\u0301";              // 'e' + combining acute U+0301

console.log(a === b);           // false (binary difference)
console.log(a.normalize() === b.normalize()); // true (after
    normalization)

// Common forms: NFC (composed), NFD (decomposed)
console.log(a.normalize("NFC") === b.normalize("NFC")); // true
```

3.6.7 Splitting, joining and parsing

Listing 3.21: Split, join and parse examples

```
// split + trim common pattern
let csv = "a, b , c ,d";
let items = csv.split(",").map(x => x.trim());
console.log(items); // ["a","b","c","d"]

// split limiting results
console.log("a,b,c,d".split(",", 2)); // ["a","b"]

// join to build CSV safely (simple)
console.log(items.join(",")); // "a,b,c,d"

// Parsing simple key=value pairs
function parseKV(str) {
  return Object.fromEntries(
    str.split(";").map(p => p.split("=").map(s => s.trim()))
  );
}
```

```
console.log(parseKV("k1=v1; k2 = v2")); // {k1: "v1", k2: "v2"}
```

3.6.8 Escaping, encoding and security

Listing 3.22: Encoding and escaping examples

```
// encode/decode URIs
let url = "https://example.com/search?q=hello world";
console.log(encodeURI(url));              // spaces and special chars
   preserved safely
console.log(encodeURIComponent("hello world")); // "hello%20world"

// Escaping HTML (simple sanitizer example - prefer secure libraries in
    prod)
function escapeHtml(str) {
  return str.replaceAll("&", "&amp;")
            .replaceAll("<", "&lt;")
            .replaceAll(">", "&gt;")
            .replaceAll('"', "&quot;")
            .replaceAll("'", "&#39;");
}
console.log(escapeHtml("<div>O'Reilly & Co</div>"));

// Avoid eval on strings. Prefer JSON.parse for JSON data
let jsonStr = '{"name":"Alice"}';
let obj = JSON.parse(jsonStr);
console.log(obj.name); // "Alice"
```

3.6.9 Performance tips and best practices

- **Avoid repeated '+' concatenation** in large loops; use `Array.push` + `join` or an efficient builder.
- **Prefer built-in methods** implemented in native code (e.g., `replace`, `slice`, `indexOf`) over manual loops where possible.
- **Be Unicode-aware**: operations like substring on surrogate pairs can break characters — use code-point aware iteration when necessary.
- **Use regular expressions carefully**: precompile / reuse regex objects when performing many matches to avoid re-parsing cost.
- **Use 'String.prototype.includes' for existence checks** instead of `indexOf` !== -1 — clearer intent and sometimes faster on modern engines.

Listing 3.23: Concatenation performance pattern

```
// Bad: repeated concatenation in loop
let big = "";
for (let i = 0; i < 10000; i++) {
  big += "line " + i + "\n"; // creates many intermediate strings
}

// Better: accumulate and join
let parts = [];
for (let i = 0; i < 10000; i++) {
  parts.push("line " + i + "\n");
}
let bigEfficient = parts.join("");
```

3.6.10 Useful advanced examples and patterns ›

Listing 3.24: Common utilities and patterns

```
// 1) Title-case a sentence (basic)
function titleCase(s) {
  return s.toLowerCase().split(" ").map(w => w.charAt(0).toUpperCase()
     + w.slice(1)).join(" ");
}
console.log(titleCase("javaScript string utilities")); // "Javascript
   String Utilities"

// 2) Safe truncation by code points (avoid breaking emoji)
function truncateByCodePoints(str, maxChars) {
  let arr = [...str]; // code point safe
  return arr.length > maxChars ? arr.slice(0, maxChars).join("") + "…"
     : str;
}
console.log(truncateByCodePoints("Hello   ", 8));

// 3) Highlight search term (case-insensitive, preserve characters)
function highlight(text, term) {
  let regex = new RegExp(`(${term.replace(/[.*+?^${}()|[\]\\]/g, "\\$
     &")})`, "ig");
  return text.replace(regex, "<mark>$1</mark>");
}
console.log(highlight("This is a test. Testing 123.", "test"));
```

3.6.11 String Methods and Manipulation ›

Table 3.7: Essential JavaScript String Methods

Method	Purpose	Example
`charAt()`	Get character at index	`str.charAt(0); // first character`
`charCodeAt()`	Get Unicode of character	`str.charCodeAt(0); // Unicode value`
`concat()`	Join strings together	`str1.concat(str2); // "hello" + "world"`
`indexOf()`	Find first occurrence	`str.indexOf("abc"); // position or -1`
`lastIndexOf()`	Find last occurrence	`str.lastIndexOf("abc"); // last position`
`slice()`	Extract part of string	`str.slice(0, 5); // characters 0-4`
`substring()`	Extract between indices	`str.substring(2, 7); // similar to slice`
`substr()`	Extract from start+length	`str.substr(2, 5); // deprecated`
`split()`	Split into array	`str.split(","); // split by comma`
`replace()`	Replace text	`str.replace("old", "new"); // first occurrence`
`replaceAll()`	Replace all occurrences	`str.replaceAll("old", "new"); // all occurrences`
`toUpperCase()`	Convert to uppercase	`str.toUpperCase(); // "HELLO"`
`toLowerCase()`	Convert to lowercase	`str.toLowerCase(); // "hello"`
`trim()`	Remove whitespace	`str.trim(); // removes start/end spaces`
`trimStart()`	Remove leading whitespace	`str.trimStart(); // removes start spaces`
`trimEnd()`	Remove trailing whitespace	`str.trimEnd(); // removes end spaces`
`padStart()`	Pad beginning	`str.padStart(10, "0"); // "000hello"`
`padEnd()`	Pad end	`str.padEnd(10, "."); // "hello....."`
`startsWith()`	Check if starts with	`str.startsWith("He"); // true/false`
`endsWith()`	Check if ends with	`str.endsWith("lo"); // true/false`
`includes()`	Check if contains	`str.includes("ell"); // true/false`
`repeat()`	Repeat string	`str.repeat(3); // "hellohellohello"`

3.6.12 Summary — Cheatsheet and pitfalls

- **Immutability:** String methods return new strings.
- **UTF-16 vs code points:** Use spread or `Array.from` for code-point aware iteration.
- **Normalization:** Use `normalize()` before comparisons when input may contain combining marks.
- **Regex:** Use `g` for global, `i` for case-insensitive, `u` for Unicode, and `m` for multiline.
- **Security:** Escape user input for HTML and URLs; never use `eval` on untrusted strings.
- **Performance:** Avoid heavy concatenation in loops; reuse regex; use native methods.

3.7. Functions

3.7.1 Understanding Functions

Functions are the building blocks of JavaScript programs. They allow us to group statements together into reusable code units.

Benefits of Functions:

- Write code once and reuse it multiple times.
- Break programs into smaller, manageable parts.
- Improve readability and maintainability.
- Avoid repetition (DRY principle).

Key concepts include:

- **Parameters:** Named inputs a function expects.
- **Arguments:** Actual values passed during function calls.
- **Return value:** The result a function produces.
- **Scope:** The visibility of variables inside and outside functions.

3.7.2 Function Declaration

A function declaration defines a named function using the `function` keyword. It is **hoisted**, which means you can call the function before its definition.

Listing 3.25: Basic Function Declaration

```
function greet(name) {
    return "Hello, " + name;
}
console.log(greet("Alice"));    // Hello, Alice
```

Listing 3.26: Function Declaration with Math

```
function square(n) {
    return n * n;
}
console.log(square(4));    // 16
console.log(square(10));   // 100
```

Listing 3.27: Function Declaration Called Before Definition

```
// Hoisting allows this
console.log(add(5, 7));    // 12

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

3.7.3 Function Expression

A function expression creates a function and assigns it to a variable. Unlike declarations, function expressions are **not hoisted**.

Listing 3.28: Simple Function Expression

```
let sayGoodbye = function(name) {
    return "Goodbye, " + name;
};
console.log(sayGoodbye("Bob"));  // Goodbye, Bob
```

Listing 3.29: Expression Stored in Constant

```
const multiply = function(a, b) {
    return a * b;
};
console.log(multiply(3, 4));  // 12
```

Listing 3.30: Anonymous Expression in Callback

```
let numbers = [1, 2, 3];
let doubled = numbers.map(function(x) {
    return x * 2;
});
console.log(doubled);  // [2, 4, 6]
```

3.7.4 Arrow Functions

Arrow functions (introduced in ES6) provide a shorter syntax. They do not have their own `this`; instead, they use the `this` from the outer scope.

Listing 3.31: Basic Arrow Function

```
const square = (x) => x * x;
console.log(square(5));  // 25
```

Listing 3.32: Arrow Function with Multiple Parameters

```
const add = (a, b) => a + b;
console.log(add(10, 20));  // 30
```

Listing 3.33: Arrow Function as Callback

```
let numbers = [1, 2, 3];
let doubled = numbers.map(n => n * 2);
console.log(doubled);  // [2, 4, 6]
```

3.7.5 Anonymous Functions

Anonymous functions are unnamed functions. They are often used as arguments for callbacks.

Listing 3.34: Anonymous Function in setTimeout

```
setTimeout(function() {
    console.log("Runs after 1 second");
}, 1000);
```

Listing 3.35: Anonymous Function in Array.map

```
let nums = [1, 2, 3];
let squared = nums.map(function(n) {
    return n * n;
});
console.log(squared); // [1, 4, 9]
```

Listing 3.36: Anonymous Function Inside Event Listener

```
document.addEventListener("click", function() {
    console.log("Document clicked!");
});
```

3.7.6 Named Function Expressions

A named function expression assigns a function with a name to a variable. The name helps in recursion and debugging.

Listing 3.37: Basic Named Function Expression

```
const factorial = function fact(n) {
    return n <= 1 ? 1 : n * fact(n - 1);
};
console.log(factorial(5));  // 120
```

Listing 3.38: Named Expression in setTimeout

```
setTimeout(function showMessage() {
    console.log("Hello after 2s");
}, 2000);
```

Listing 3.39: Named Expression for Debugging

```
let divide = function safeDivide(a, b) {
    if (b === 0) throw new Error("Division by zero");
    return a / b;
};
console.log(divide(10, 2));  // 5
```

3.7.7 Constructor Functions

Constructor functions are used to create multiple objects of the same type. By convention, their names start with a capital letter and they are called with the `new` keyword.

Listing 3.40: Basic Constructor Function

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
let p1 = new Person("Alice", 25);
let p2 = new Person("Bob", 30);
console.log(p1.name, p1.age);  // Alice 25
console.log(p2.name, p2.age);  // Bob 30
```

Listing 3.41: Constructor with Method

```
function Car(model, year) {
    this.model = model;
    this.year = year;
    this.drive = function() {
        return this.model + " is driving";
    };
}
let car = new Car("Toyota", 2020);
console.log(car.drive());  // Toyota is driving
```

Listing 3.42: Using Prototype with Constructor

```
function Animal(type) {
    this.type = type;
}
Animal.prototype.speak = function() {
    return this.type + " makes a sound";
};
let dog = new Animal("Dog");
console.log(dog.speak());  // Dog makes a sound
```

3.7.8 Generator Functions

Generator functions are declared with `function*`. They return an iterator object and can yield multiple values one by one.

Listing 3.43: Simple Generator

```
function* countToThree() {
    yield 1;
    yield 2;
    yield 3;
}
let it = countToThree();
console.log(it.next().value); // 1
console.log(it.next().value); // 2
console.log(it.next().value); // 3
```

Listing 3.44: Infinite Generator

```
function* naturalNumbers() {
    let n = 1;
    while(true) {
        yield n++;
    }
}
let gen = naturalNumbers();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

Listing 3.45: Generator with Loop

```
function* fruits() {
    yield "Apple";
```

```
    yield "Banana";
    yield "Mango";
}
for (let fruit of fruits()) {
    console.log(fruit);
}
// Apple, Banana, Mango
```

3.7.9 Async Functions

Async functions simplify working with Promises. They always return a Promise and allow the use of await.

Listing 3.46: Basic Async Function

```
async function greet() {
    return "Hello Async";
}
greet().then(console.log);  // Hello Async
```

Listing 3.47: Async with Await

```
function delay(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

async function run() {
    console.log("Start");
    await delay(1000);
    console.log("After 1 second");
}
run();
```

Listing 3.48: Fetching Data with Async/Await

```
async function fetchData() {
    let response = await fetch("https://jsonplaceholder.typicode.com/
       todos/1");
    let data = await response.json();
    console.log(data);
}
fetchData();
```

3.7.10 Immediately Invoked Function Expressions (IIFE)

An IIFE is a function that runs immediately after being defined. It is useful for avoiding global variable pollution.

Listing 3.49: Simple IIFE

```
(function() {
    console.log("IIFE runs immediately");
})();
```

Listing 3.50: IIFE with Parameters

```
(function(name) {
    console.log("Hello, " + name);
})("Alice");  // Hello, Alice
```

Listing 3.51: IIFE Returning a Value

```
let result = (function(a, b) {
    return a + b;
})(5, 7);
console.log(result);  // 12
```

3.7.11 Methods in Objects

Methods are functions that belong to an object. They provide behavior to objects and use `this` to access properties.

Listing 3.52: Object Method

```
let user = {
    name: "Alice",
    greet: function() {
        return "Hello, " + this.name;
    }
};
console.log(user.greet());  // Hello, Alice
```

Listing 3.53: Shorthand Method Syntax

```
let calculator = {
    add(a, b) { return a + b; },
```

```
    multiply(a, b) { return a * b; }
};
console.log(calculator.add(5, 10));        // 15
console.log(calculator.multiply(3, 4));    // 12
```

Listing 3.54: Using this in Methods

```
let car = {
    brand: "Tesla",
    show() {
        console.log("Car brand: " + this.brand);
    }
};
car.show();  // Car brand: Tesla
```

3.7.12 Getters and Setters

Getters retrieve property values, while setters change them. They allow controlled access to object properties.

Listing 3.55: Basic Getter and Setter

```
let user = {
    firstName: "Alice",
    lastName: "Smith",
    get fullName() {
        return this.firstName + " " + this.lastName;
    },
    set fullName(name) {
        [this.firstName, this.lastName] = name.split(" ");
    }
};
console.log(user.fullName);    // Alice Smith
user.fullName = "Bob Brown";
console.log(user.fullName);    // Bob Brown
```

Listing 3.56: Getter for Computed Value

```
let rectangle = {
    length: 5,
    width: 3,
    get area() {
        return this.length * this.width;
```

```
    }
};
console.log(rectangle.area); // 15
```

Listing 3.57: Setter with Validation

```
let account = {
    balance: 1000,
    set deposit(amount) {
        if (amount > 0) this.balance += amount;
    }
};
account.deposit = 500;
console.log(account.balance); // 1500
```

3.7.13 Closures

Closures occur when an inner function remembers variables from its outer function even after the outer function has finished execution.

Listing 3.58: Basic Closure Example

```
function outer() {
    let message = "Hello from outer";
    return function inner() {
        return message;
    };
}
let fn = outer();
console.log(fn());  // Hello from outer
```

Listing 3.59: Counter Using Closure

```
function createCounter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}
let counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Listing 3.60: Module Pattern with Closure

```
let calculator = (function() {
    let result = 0;
    return {
        add(x) { result += x; return this; },
        subtract(x) { result -= x; return this; },
        getResult() { return result; }
    };
})();
calculator.add(10).subtract(3);
console.log(calculator.getResult());  // 7
```

3.8. Methods and Objects

3.8.1 Understanding Objects

Objects are collections of key-value pairs that represent real-world entities or abstract concepts. Objects can contain properties (data) and methods (functions). They are fundamental to JavaScript and form the basis of object-oriented programming.

Object Characteristics:

- **Properties:** Variables that belong to the object
- **Methods:** Functions that belong to the object
- **Dynamic:** Properties can be added, modified, or removed
- **Reference type:** Objects are passed by reference

3.8.2 Object Creation and Manipulation

Listing 3.61: Object Creation and Basic Operations

```
// Object literal notation (most common)
let person = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
    email: "john.doe@example.com",
    isEmployed: true,
    hobbies: ["reading", "gaming", "cooking"]
};

// Accessing object properties
console.log(person.firstName);        // "John" (dot notation)
console.log(person["lastName"]);      // "Doe" (bracket notation)
```

```
console.log(person.age);              // 30

// Adding new properties
person.phone = "123-456-7890";
person["address"] = "123 Main St";

// Modifying existing properties
person.age = 31;
person.email = "john.new@example.com";

// Deleting properties
delete person.isEmployed;

// Object constructor function
function Person(firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.getFullName = function() {
        return this.firstName + " " + this.lastName;
    };
}

let person1 = new Person("Alice", "Smith", 25);
let person2 = new Person("Bob", "Johnson", 35);

console.log(person1.getFullName());   // "Alice Smith"
console.log(person2.getFullName());   // "Bob Johnson"

// Object.create() method
let personPrototype = {
    greet: function() {
        return `Hello, I'm ${this.name}`;
    }
};

let student = Object.create(personPrototype);
student.name = "Charlie";
student.grade = "A";
console.log(student.greet());         // "Hello, I'm Charlie"

// ES6 Class syntax (modern approach)
class Modern Person {
    constructor(firstName, lastName, age) {
        this.firstName = firstName;
        this.lastName = lastName;
```

```
        this.age = age;
    }

    getFullName() {
        return `${this.firstName} ${this.lastName}`;
    }

    introduce() {
        return `Hi, I'm ${this.getFullName()} and I'm ${this.age} years
            old.`;
    }
}

let person3 = new ModernPerson("Diana", "Wilson", 28);
console.log(person3.introduce());
```

3.8.3 Object Methods and this Keyword

Listing 3.62: Object Methods and 'this' Keyword

```
let calculator = {
    result: 0,

    // Method definitions
    add: function(number) {
        this.result += number;
        return this;  // Return this for method chaining
    },

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

    multiply: function(number) {
        this.result *= number;
        return this;
    },

    divide: function(number) {
        if (number !== 0) {
            this.result /= number;
        } else {
            console.log("Cannot divide by zero!");
```

```
        }
        return this;
    },

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

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

    // ES6 method shorthand
    display() {
        console.log(`Current result: ${this.result}`);
        return this;
    }
};

// Method chaining
calculator
    .clear()
    .add(10)
    .multiply(3)
    .subtract(5)
    .divide(5)
    .display()          // Current result: 5
    .getResult();       // 5

// Bank account example
let bankAccount = {
    accountNumber: "123456789",
    accountHolder: "John Doe",
    balance: 1000,

    deposit: function(amount) {
        if (amount > 0) {
            this.balance += amount;
            console.log(`Deposited ${amount}. New balance: ${this.
               balance}`);
        } else {
            console.log("Deposit amount must be positive.");
        }
        return this;
    },
```

```
    withdraw: function(amount) {
        if (amount > 0 && amount <= this.balance) {
            this.balance -= amount;
            console.log(`Withdrew ${amount}. New balance: ${this.
               balance}`);
        } else if (amount > this.balance) {
            console.log("Insufficient funds.");
        } else {
            console.log("Withdrawal amount must be positive.");
        }
        return this;
    },

    getBalance: function() {
        return this.balance;
    },

    getAccountInfo: function() {
        return {
            number: this.accountNumber,
            holder: this.accountHolder,
            balance: this.balance
        };
    }
};

// Using the bank account
bankAccount
    .deposit(500)        // Deposited $500. New balance: $1500
    .withdraw(200)       // Withdrew $200. New balance: $1300
    .withdraw(2000);     // Insufficient funds.

console.log(bankAccount.getAccountInfo());
```

3.8.4 Built-in Object Methods

Table 3.8: Essential JavaScript Object Methods

Method	Purpose	Example
`Object.keys()`	Get array of property names	`Object.keys(obj); // ["prop1", "prop2"]`
`Object.values()`	Get array of property values	`Object.values(obj); // [value1, value2]`
`Object.entries()`	Get array of [key, value] pairs	`Object.entries(obj); // [["prop1", val1]]`

Continued on next page

Table 3.8 continued from previous page

Method	Purpose	Example
`Object.assign()`	Copy properties to target	`Object.assign(target, source1, source2)`
`Object.freeze()`	Prevent modifications	`Object.freeze(obj); // obj becomes immutable`
`Object.seal()`	Prevent adding/removing props	`Object.seal(obj); // existing props can change`
`hasOwnProperty()`	Check if property exists	`obj.hasOwnProperty("prop"); // true/false`
`in` operator	Check property existence	`"prop" in obj; // true/false`
`typeof`	Get object type	`typeof obj; // "object"`
`instanceof`	Check object type	`obj instanceof Array; // true/false`

Listing 3.63: Working with Built-in Object Methods

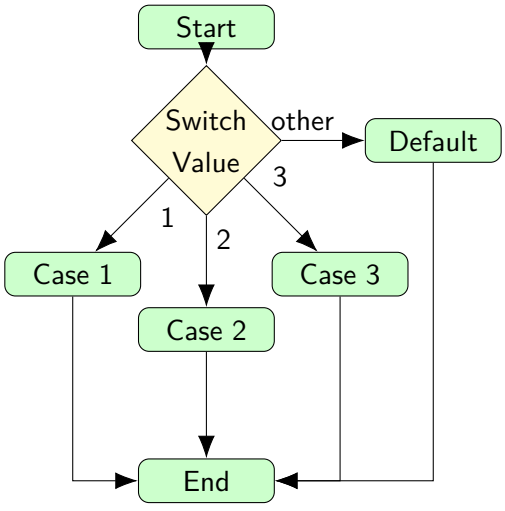
```
let student = {
    name: "Alice Johnson",
    age: 22,
    major: "Computer Science",
    gpa: 3.8,
    courses: ["Math", "Physics", "Programming"]
};

// Getting object information
console.log(Object.keys(student));      // ["name", "age", "major", "gpa
    ", "courses"]
console.log(Object.values(student));    // ["Alice Johnson", 22, "
    Computer Science", 3.8, Array]
console.log(Object.entries(student));   // [["name", "Alice Johnson"],
    ["age", 22], ...]

// Checking properties
console.log(student.hasOwnProperty("name"));     // true
console.log(student.hasOwnProperty("height"));   // false
console.log("age" in student);                   // true
console.log("toString" in student);              // true (inherited)

// Copying objects
let studentCopy = Object.assign({}, student);    // Shallow copy
let anotherCopy = {...student};                  // Spread operator (ES6
    )

// Merging objects
let additionalInfo = {
    email: "alice@university.edu",
    year: "Senior"
```

```
};
let completeStudent = Object.assign({}, student, additionalInfo);

// Iterating over objects
for (let key in student) {
    if (student.hasOwnProperty(key)) {
        console.log(`${key}: ${student[key]}`);
    }
}

// Modern iteration methods
Object.keys(student).forEach(key => {
    console.log(`${key}: ${student[key]}`);
});

// Converting object to array format for processing
let studentArray = Object.entries(student).map(([key, value]) => {
    return {property: key, value: value};
});

// Object destructuring
let {name, age, major, ...rest} = student;
console.log(name);  // "Alice Johnson"
console.log(age);   // 22
console.log(rest);  // {gpa: 3.8, courses: [...]}
\end{lstlisting>

% ============================
% DECISIONS AND CONTROL STRUCTURES
% ============================
\mysection{Decisions and Control Structures}

\mysubsection{Conditional Statements}

Conditional statements allow your program to make decisions based on
   different conditions. They control which code blocks execute based
   on whether certain conditions are true or false.

\mysubsubsection{if Statement}

\begin{center}
\begin{tikzpicture}[node distance = 2cm, auto]
    % Define styles
    \tikzstyle{decision} = [diamond, draw, fill=blue!20, text width=4.5
       em, text badly centered, inner sep=0pt]
    \tikzstyle{process} = [rectangle, draw, fill=green!20, text width=5
```

```
        em, text centered, rounded corners, minimum height=2em]
    \tikzstyle{line} = [draw, -{Latex[length=3mm]}]


    % Place nodes
    \node [process] (start) {Start};
    \node [decision, below of=start] (condition) {Condition True?};
    \node [process, below of=condition, node distance=3cm] (action) {
       Execute Code Block};
    \node [process, right of=condition, node distance=4cm] (skip) {Skip
        Code Block};
    \node [process, below of=action] (end) {Continue};

    % Draw edges
    \path [line] (start) -- (condition);
    \path [line] (condition) -- node [near start] {yes} (action);
    \path [line] (condition) -- node [near start] {no} (skip);
    \path [line] (action) -- (end);
    \path [line] (skip) |- (end);
\end{tikzpicture}
\end{center}

\begin{lstlisting}[style=jsstyle,caption={if Statement Examples},
   language=JavaScript]
let age = 20;
let hasLicense = true;
let weather = "sunny";

// Simple if statement
if (age >= 18) {
    console.log("You are an adult.");
}

// if-else statement
if (hasLicense) {
    console.log("You can drive.");
} else {
    console.log("You cannot drive without a license.");
}

// if-else if-else statement
if (age < 13) {
    console.log("You are a child.");
} else if (age < 20) {
    console.log("You are a teenager.");
} else if (age < 65) {
```

```
    console.log("You are an adult.");
} else {
    console.log("You are a senior.");
}

// Complex conditions with logical operators
if (age >= 16 && hasLicense) {
    console.log("You can drive legally.");
}

if (weather === "sunny" || weather === "cloudy") {
    console.log("Good day for outdoor activities.");
}

// Nested if statements
if (age >= 18) {
    if (hasLicense) {
        console.log("You can rent a car.");
    } else {
        console.log("You need a license to rent a car.");
    }
}
```

3.8.4.1 *switch Statement*



Listing 3.64: switch Statement Examples

```
let dayOfWeek = 3;
let dayName;

// Basic switch statement
```

```
switch (dayOfWeek) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    case 4:
        dayName = "Thursday";
        break;
    case 5:
        dayName = "Friday";
        break;
    case 6:
        dayName = "Saturday";
        break;
    case 7:
        dayName = "Sunday";
        break;
    default:
        dayName = "Invalid day";
}
console.log(dayName); // "Wednesday"

// Switch with multiple cases
let grade = "B";
let message;

switch (grade) {
    case "A":
    case "A+":
        message = "Excellent work!";
        break;
    case "B":
    case "B+":
        message = "Good job!";
        break;
    case "C":
        message = "Satisfactory.";
        break;
    case "D":
        message = "Needs improvement.";
        break;
```

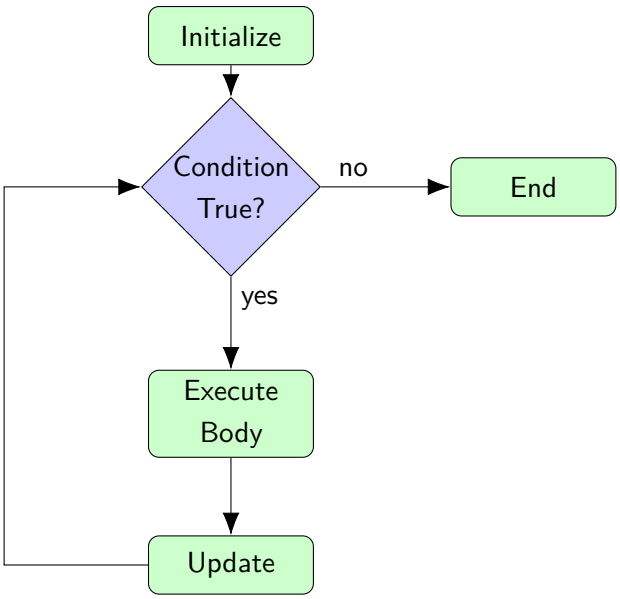
```
    case "F":
        message = "Failed.";
        break;
    default:
        message = "Invalid grade.";
}
console.log(message); // "Good job!"

// Switch with expressions
function getSeasonByMonth(month) {
    switch (month) {
        case 12:
        case 1:
        case 2:
            return "Winter";
        case 3:
        case 4:
        case 5:
            return "Spring";
        case 6:
        case 7:
        case 8:
            return "Summer";
        case 9:
        case 10:
        case 11:
            return "Fall";
        default:
            return "Invalid month";
    }
}

console.log(getSeasonByMonth(7));  // "Summer"
console.log(getSeasonByMonth(13)); // "Invalid month"
```

3.9. Loops

3.9.1 Understanding Loops

Loops allow you to execute code repeatedly until a certain condition is met. They are essential for processing arrays, generating sequences, and automating repetitive tasks.

3.9.1.1 *for Loop*

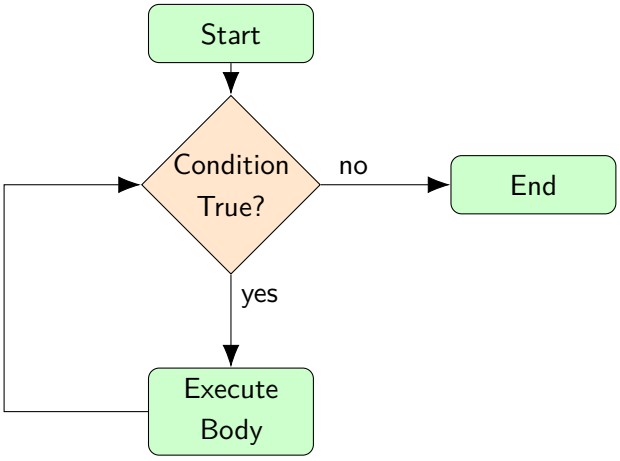


Listing 3.65: for Loop Examples

```
// Basic for loop
for (let i = 0; i < 5; i++) {
    console.log("Count: " + i);
}
// Output: Count: 0, Count: 1, Count: 2, Count: 3, Count: 4

// Looping through an array
let fruits = ["apple", "banana", "orange", "grape"];

for (let i = 0; i < fruits.length; i++) {
    console.log(`Fruit ${i + 1}: ${fruits[i]}`);
}

// Reverse loop
for (let i = fruits.length - 1; i >= 0; i--) {
    console.log(`Reverse order: ${fruits[i]}`);
}

// Loop with step increment
for (let i = 0; i <= 20; i += 2) {
    console.log("Even number: " + i);
}

// Nested for loops - multiplication table
console.log("Multiplication Table:");
for (let i = 1; i <= 5; i++) {
    let row = "";
    for (let j = 1; j <= 5; j++) {
```

```
            row += (i * j).toString().padStart(3, " ");
        }
        console.log(row);
    }
```

3.9.1.2 *while Loop*

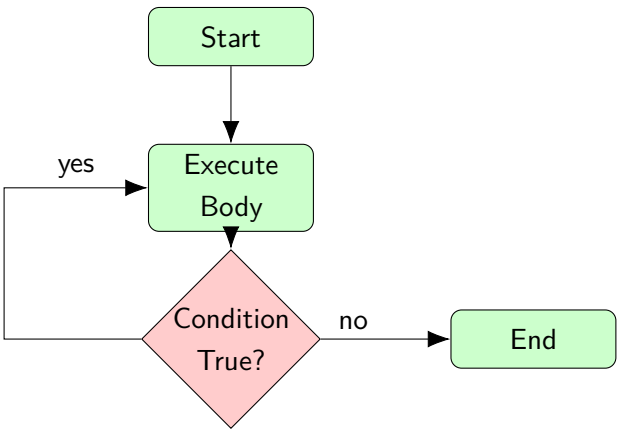


Listing 3.66: while Loop Examples

```
// Basic while loop
let count = 0;
while (count < 5) {
    console.log("Count: " + count);
    count++; // Important: update the condition variable
}

// While loop with user input simulation
let userInput = "";
let attempts = 0;
let maxAttempts = 3;

while (userInput !== "quit" && attempts < maxAttempts) {
    // Simulate getting user input
    userInput = ["hello", "world", "quit"][attempts];
    console.log(`Attempt ${attempts + 1}: ${userInput}`);
    attempts++;
}

// While loop for finding elements
let numbers = [2, 4, 7, 8, 10, 13, 16];
let target = 8;
let index = 0;
```

```
let found = false;

while (index < numbers.length && !found) {
    if (numbers[index] === target) {
        found = true;
        console.log(`Found ${target} at index ${index}`);
    } else {
        index++;
    }
}

if (!found) {
    console.log(`${target} not found in array`);
}

// Infinite loop prevention example
let safety = 0;
let condition = true;

while (condition && safety < 1000) {
    // Your loop logic here
    safety++; // Safety counter

    // Some condition to eventually break the loop
    if (safety > 10) {
        condition = false;
    }
}
```

3.9.1.3 *do-while Loop*



Listing 3.67: do-while Loop Examples

```
// Basic do-while loop - executes at least once
let userChoice;
do {
    // Simulate menu selection
    console.log("\nMenu:");
    console.log("1. View Profile");
    console.log("2. Edit Settings");
    console.log("3. Exit");

    // Simulate user input
    userChoice = Math.floor(Math.random() * 4) + 1; // Random choice
    console.log(`You selected: ${userChoice}`);

    switch (userChoice) {
        case 1:
            console.log("Viewing profile...");
            break;
        case 2:
            console.log("Editing settings...");
            break;
        case 3:
            console.log("Goodbye!");
            break;
        default:
            console.log("Invalid choice. Please try again.");
    }
} while (userChoice !== 3);

// Password validation example
let password;
let isValidPassword = false;

do {
    // Simulate getting password input
    password = "weak"; // This would come from user input

    // Validate password
    if (password.length >= 8) {
        isValidPassword = true;
        console.log("Password accepted!");
    } else {
        console.log("Password too short. Must be at least 8 characters.
            ");
        // In real scenario, prompt user again
        password = "strongPassword123"; // Simulate new input
    }
```

```
} while (!isValidPassword);

// Dice rolling game
let diceRoll;
let rollCount = 0;

console.log("Rolling dice until we get a 6...");
do {
    diceRoll = Math.floor(Math.random() * 6) + 1;
    rollCount++;
    console.log(`Roll ${rollCount}: ${diceRoll}`);
} while (diceRoll !== 6);

console.log(`Got a 6 after ${rollCount} rolls!`);
```

3.9.1.4 *for...in and for...of Loops*

Listing 3.68: for...in and for...of Loop Examples

```
let student = {
    name: "Alice",
    age: 22,
    major: "Computer Science",
    gpa: 3.8
};

let courses = ["Math", "Physics", "Programming", "Database"];

// for...in loop - iterates over object properties
console.log("Student properties:");
for (let property in student) {
    console.log(`${property}: ${student[property]}`);
}

// for...in with arrays (gets indices)
console.log("\nArray indices:");
for (let index in courses) {
    console.log(`Index ${index}: ${courses[index]}`);
}

// for...of loop - iterates over array values
console.log("\nCourse list:");
for (let course of courses) {
    console.log(`Course: ${course}`);
}
```

```
// for...of with strings
let message = "Hello";
console.log("\nCharacters in message:");
for (let char of message) {
    console.log(char);
}

// Advanced for...of with entries
console.log("\nCourses with indices:");
for (let [index, course] of courses.entries()) {
    console.log(`${index + 1}. ${course}`);
}

// for...of with Map
let grades = new Map([
    ["Math", "A"],
    ["Physics", "B+"],
    ["Programming", "A+"]
]);

console.log("\nGrades:");
for (let [subject, grade] of grades) {
    console.log(`${subject}: ${grade}`);
}
```

3.9.2 Loop Control Statements

Table 3.9: JavaScript Loop Control Statements

Statement	Purpose	Example
break	Exit loop completely	`if (condition) break;`
continue	Skip current iteration	`if (condition) continue;`
return	Exit function (and loop)	`if (found) return value;`
label:	Label for break/continue	`outer: for(...) { inner: for(...) break outer; }`

Listing 3.69: Loop Control Examples

```
// break statement - exit loop early
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let target = 5;

console.log("Finding target number:");
for (let i = 0; i < numbers.length; i++) {
```

```
    console.log(`Checking: ${numbers[i]}`);
    if (numbers[i] === target) {
        console.log(`Found target ${target}!`);
        break; // Exit the loop
    }
}

// continue statement - skip current iteration
console.log("\nEven numbers only:");
for (let i = 1; i <= 10; i++) {
    if (i % 2 !== 0) {
        continue; // Skip odd numbers
    }
    console.log(i);
}

// Nested loops with labeled break
console.log("\nSearching in 2D array:");
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

let searchValue = 6;
let found = false;

outerLoop: for (let row = 0; row < matrix.length; row++) {
    for (let col = 0; col < matrix[row].length; col++) {
        console.log(`Checking position [${row}][${col}]: ${matrix[row][
            col]}`);
        if (matrix[row][col] === searchValue) {
            console.log(`Found ${searchValue} at position [${row}][${
                col}]`);
            found = true;
            break outerLoop; // Break out of both loops
        }
    }
}

if (!found) {
    console.log(`${searchValue} not found in matrix`);
}

// Practical example: Input validation
function validateInput(inputs) {
```

```
    let validInputs = [];

    for (let input of inputs) {
        // Skip empty or null inputs
        if (!input || input.trim() === "") {
            console.log("Skipping empty input");
            continue;
        }

        // Stop processing if we encounter "STOP"
        if (input.toUpperCase() === "STOP") {
            console.log("Stop command encountered");
            break;
        }

        // Process valid input
        validInputs.push(input.trim());
    }

    return validInputs;
}

let testInputs = ["hello", "", "world", null, "STOP", "ignored"];
let result = validateInput(testInputs);
console.log("Valid inputs:", result); // ["hello", "world"]
```

3.9.3 Loop Performance and Best Practices

Listing 3.70: Loop Best Practices

```
// Performance optimization examples
let largeArray = new Array(1000000).fill(0).map((_, i) => i);

// Good: Cache array length
console.time("Cached length");
for (let i = 0, len = largeArray.length; i < len; i++) {
    // Process element
}
console.timeEnd("Cached length");

// Less efficient: Access length property each iteration
console.time("Non-cached length");
for (let i = 0; i < largeArray.length; i++) {
    // Process element
```

```
}
console.timeEnd("Non-cached length");

// Modern alternatives - often more readable and performant
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// forEach - good for simple operations
console.log("Using forEach:");
data.forEach(num => {
    if (num % 2 === 0) {
        console.log(`Even: ${num}`);
    }
});

// map - good for transformations
let doubled = data.map(num => num * 2);
console.log("Doubled:", doubled);

// filter - good for selecting elements
let evens = data.filter(num => num % 2 === 0);
console.log("Even numbers:", evens);

// reduce - good for accumulation
let sum = data.reduce((total, num) => total + num, 0);
console.log("Sum:", sum);

// find - good for searching
let firstEven = data.find(num => num % 2 === 0);
console.log("First even:", firstEven);

// some/every - good for testing conditions
let hasEven = data.some(num => num % 2 === 0);
let allPositive = data.every(num => num > 0);
console.log("Has even number:", hasEven);
console.log("All positive:", allPositive);

// Avoiding common pitfalls
console.log("\nCommon Loop Pitfalls:");

// Problem: Infinite loop
// while (true) {
//     console.log("This would run forever!");
// }

// Solution: Always ensure loop condition can become false
let counter = 0;
```

```
while (counter < 5) {
    console.log(`Safe loop: ${counter}`);
    counter++; // Always update the condition variable
}

// Problem: Modifying array while iterating
let items = ["a", "b", "c", "d", "e"];

// Wrong way - skips elements when removing
// for (let i = 0; i < items.length; i++) {
//     if (items[i] === "b" || items[i] === "d") {
//         items.splice(i, 1);
//     }
// }

// Right way - iterate backwards when removing
for (let i = items.length - 1; i >= 0; i--) {
    if (items[i] === "b" || items[i] === "d") {
        items.splice(i, 1);
    }
}

// Or better - use filter
let filteredItems = items.filter(item => item !== "b" && item !== "d");
console.log("Filtered items:", filteredItems);
```

Chapter Summary: JavaScript Fundamentals

In this comprehensive chapter, we've covered the essential building blocks of JavaScript programming:

Core Concepts Covered:

- **Statements & Comments:** Understanding code structure and documentation
- **Variables:** Storage containers using `let`, `const`, and `var`
- **Data Types:** Primitive types (number, string, boolean, null, undefined) and their operations
- **Arrays:** Ordered collections with powerful built-in methods
- **Strings:** Text manipulation and processing techniques
- **Functions:** Reusable code blocks with parameters and return values
- **Objects & Methods:** Key-value pairs and object-oriented programming concepts
- **Control Structures:** Decision making with `if/else` and `switch`
- **Loops:** Repetitive execution with `for`, `while`, and modern iteration methods

Key Skills Developed:

- Writing clean, well-commented code
- Choosing appropriate data types and structures
- Creating and using functions effectively
- Implementing conditional logic and loops
- Working with arrays and objects
- Understanding scope and closures

Next Chapter Preview: We'll explore advanced JavaScript concepts including ES6+ features, asynchronous programming, error handling, and modern development patterns that will prepare you for building complex web applications.

3.10. UNIT I – JavaScript Basics – Questions and Answers

3.10.1 Explain different types of JavaScript statements with suitable examp

JavaScript statements are instructions that tell the browser what to do. Important types are:

- **Declaration Statements** – used to declare variables.
- **Assignment Statements** – assign values to variables.
- **Conditional Statements** – execute code based on a condition.
- **Looping Statements** – repeat a block of code multiple times.
- **Function Statements** – group reusable blocks of code.

```
// Declaration
let x;

// Assignment
x = 20;
```

```
// Conditional
if (x > 10) console.log("Greater");

// Loop
for (let i = 0; i < 5; i++) console.log(i);

// Function
function greet() { console.log("Hello"); }
```

3.10.2 What are variables in JavaScript? Explain var, let, and const with ex

Variables are containers used to store data. JavaScript provides three keywords:

- **var** – Function-scoped, can be re-declared, older style.
- **let** – Block-scoped, cannot be redeclared in the same scope.
- **const** – Block-scoped, value cannot be reassigned.

```
var a = 10;
var a = 20;   // allowed

let b = 10;
// let b = 20; // Error

const c = 30;
// c = 40;     // Error
```

3.10.3 Explain JavaScript data types with examples.

JavaScript supports two categories of data types:

- **Primitive Data Types**
  - String, Number, Boolean
  - Undefined, Null
  - Symbol, BigInt
- **Non-Primitive Data Types**
  - Objects, Arrays, Functions

```
// Primitive
let s = "Hello";      // String
let n = 42;           // Number
let flag = true;      // Boolean
let u;                // Undefined
let x = null;         // Null
```

```
let sym = Symbol("id");
let big = 12345678901234567890n;

// Object
let person = { name: "John", age: 25 };
```

3.10.4 Write a short note on arrays in JavaScript. Explain at least five array

An array is a data structure used to store multiple values in a single variable. Key features:

- Ordered collection of elements.
- Can hold different data types.
- Provides inbuilt methods for manipulation.

Common methods:

- **push()** – add element at the end.
- **pop()** – remove last element.
- **shift()** – remove first element.
- **unshift()** – add element at the start.
- **map()** – apply function to each element.

```
let arr = [1, 2, 3];
arr.push(4);      // [1,2,3,4]
arr.pop();        // [1,2,3]
arr.shift();      // [2,3]
arr.unshift(0);   // [0,2,3]
arr.map(x => x*2); // [0,4,6]
```

3.10.5 Explain string manipulation in JavaScript with examples›

Strings are sequences of characters. Common string operations:

- Finding length using `length`.
- Changing case using `toUpperCase()`, `toLowerCase()`.
- Checking presence of substring with `includes()`.
- Replacing parts of string with `replace()`.

```
let str = "Hello World";
console.log(str.length);
console.log(str.toUpperCase());
console.log(str.toLowerCase());
console.log(str.includes("Hello"));
console.log(str.replace("World","JS"));
```

3.10.6 Differentiate between function declaration, function expression, and a

- **Function Declaration** – hoisted, can be called before definition.
- **Function Expression** – assigned to variable, not hoisted.
- **Arrow Function** – shorter syntax, does not have its own `this`.

```
// Function Declaration
function add(a, b) { return a + b; }

// Function Expression
const addExp = function(a, b) { return a + b; };

// Arrow Function
const addArrow = (a, b) => a + b;
```

3.10.7 What are objects in JavaScript? Explain object creation and accessin

Objects store key-value pairs and allow grouping of related data.

- Can store properties (data).
- Can have methods (functions).
- Accessed using dot or bracket notation.

```
let person = {
  name: "Alice",
  age: 22,
  greet: function() { return "Hello, " + this.name; }
};

console.log(person.name);
console.log(person["age"]);
console.log(person.greet());
```

3.10.8 Explain the different decision-making statements (if...else, switch) wi

Decision-making allows execution based on conditions.

- **if...else** – executes a block based on condition.
- **switch** – selects one case among multiple options.

```
let num = 5;
if (num > 0) console.log("Positive");
else console.log("Non-positive");
```

```
let day = 2;
switch(day) {
  case 1: console.log("Monday"); break;
  case 2: console.log("Tuesday"); break;
  default: console.log("Other day");
}
```

3.10.9 Explain looping constructs in JavaScript (for, while, do-while, for...of

Loops are used to execute a block of code multiple times.

- **for** – executes a fixed number of times.
- **while** – executes while condition is true.
- **do-while** – executes at least once before checking condition.
- **for...of** – iterates over iterable objects like arrays.

```
// for
for (let i = 0; i < 3; i++) console.log(i);

// while
let j = 0;
while (j < 3) { console.log(j); j++; }

// do-while
let k = 0;
do { console.log(k); k++; } while (k < 3);

// for...of
let arr = ["a","b","c"];
for (let ch of arr) console.log(ch);
```

3.10.10 Write a JavaScript program to find the factorial of a number using a

Factorial ($n!$) is the product of numbers from 1 to n.

```
let n = 5;
let fact = 1;

for (let i = 1; i <= n; i++) {
  fact *= i;
}
console.log("Factorial:", fact); // 120
```

CHAPTER 4

4 DOM & JavaScript Events

4.1. Introduction to the DOM

The **Document Object Model (DOM)** is a programming interface provided by the browser for working with web documents. Whenever a web page is loaded in the browser, the browser converts the `HTML` (or `XML`) code into a structured representation that JavaScript programs can access and manipulate.

In simple words, the DOM is a *bridge* between the HTML document and JavaScript. It allows scripts to dynamically read, update, and change both the structure and the content of a web page while it is running.

4.1.1 What is the DOM?

- The DOM represents an HTML (or XML) document as a **tree structure**, often called the *DOM tree*.
- Each part of the document — elements, attributes, and text — is represented as a **node** in this tree.
- JavaScript can use special methods provided by the DOM API to *navigate*, *search*, *modify*, and *delete* these nodes.
- Any changes made through the DOM API are immediately reflected in the page that the user sees in the browser.

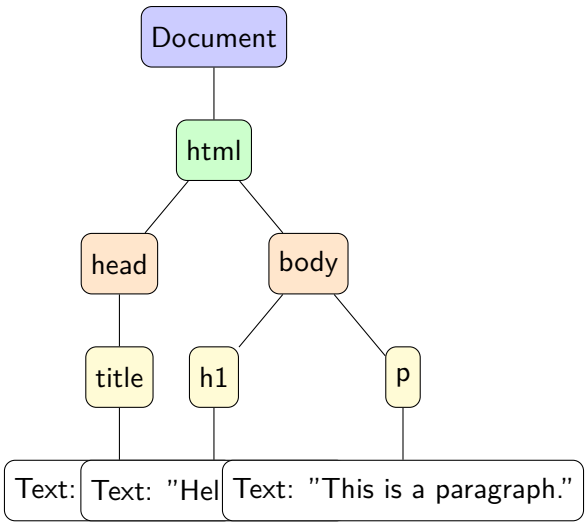
4.1.2 DOM Tree Representation

Consider a simple HTML page:

Listing 4.1: Example HTML document

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

When the browser loads this code, it creates the following DOM tree:



Here:

- The root is the **Document node**.
- The `<html>` element is the top-level element of the page.
- Inside it, we have `<head>` and `<body>` elements.
- Elements like `<title>`, `<h1>`, and `<p>` contain **text nodes**.

4.1.3 Why is the DOM Important?

The DOM is essential for modern web programming because it allows:

- **Dynamic Content:** JavaScript can insert new elements, delete old ones, or update existing text and attributes.
- **Interactivity:** Buttons, forms, and menus respond to user actions by modifying the DOM.
- **Cross-Platform Standard:** The DOM is standardized by the W3C, so scripts work across all major browsers.

4.1.4 Example: Modifying the DOM

Listing 4.2: Updating DOM content with JavaScript

```
<h1 id="heading">Welcome</h1>
<button onclick="changeText()">Change Text</button>

<script>
  function changeText() {
    // Get the DOM node by its id
    const h = document.getElementById("heading");
    // Modify the text content of the node
    h.textContent = "Hello, DOM World!";
  }
</script>
```

Explanation:

- The HTML defines a heading and a button.
- The JavaScript code selects the `<h1>` element using `document.getElementById`.
- When the button is clicked, the script changes the heading text. This demonstrates how DOM manipulations are instantly reflected on the web page.

4.1.5 Summary

- The DOM is a **tree-like representation** of a web document.
- Each element, attribute, and piece of text is a **node** in the tree.
- JavaScript can access and manipulate the DOM to build interactive, dynamic pages.
- Without the DOM, JavaScript could not modify what is displayed on the browser after the page is loaded.

Thus, the DOM is the foundation for making web applications interactive.

4.2. The HTML DOM Document Object

The **document object** is the root of the HTML DOM hierarchy. When a web page is loaded, the browser creates a **Document Object**, which becomes the "owner" of all other elements, attributes, and text nodes on the page.

- The document object represents your entire HTML page.
- All DOM access in JavaScript starts with `document`.
- You can use it to *find*, *create*, *modify*, or *delete* elements and also to attach events.

4.2.1 Finding HTML Elements

The document object provides multiple methods to locate elements:

Method	Description
`document.getElementById(id)`	Finds an element by its unique `id`.
`document.getElementsByTagName(name)`	Returns a collection of elements with the given tag name.
`document.getElementsByClassName(name)`	Returns a collection of elements with the given class name.
`document.querySelector(css)`	Returns the first element that matches a CSS selector.
`document.querySelectorAll(css)`	Returns all elements that match a CSS selector.

Listing 4.3: Finding elements with document object

```
<p id="intro">Hello</p>
<p class="msg">Welcome</p>

<script>
  let elem1 = document.getElementById("intro");
  console.log(elem1.textContent); // Hello

  let elems = document.getElementsByClassName("msg");
```

```
  console.log(elems[0].textContent); // Welcome

  let firstP = document.querySelector("p");
  console.log(firstP.textContent); // Hello
</script>
```

4.2.2 Changing HTML Elements

Once elements are selected, they can be modified using properties and methods.

Property/Method	Description
`element.innerHTML = "..."`	Change the inner HTML (markup + text) of an element.
`element.textContent = "..."`	Change only the text inside an element (safer).
`element.attribute = value`	Change an attribute value directly.
`element.setAttribute(name, value)`	Change/add an attribute.
`element.style.property = value`	Change CSS style.

Listing 4.4: Changing content and attributes

```
<h1 id="title">Old Title</h1>
<img id="pic" src="a.jpg">

<script>
  let h = document.getElementById("title");
  h.textContent = "New Title";

  let img = document.getElementById("pic");
  img.setAttribute("src", "b.jpg");
  img.style.border = "2px solid red";
</script>
```

4.2.3 Adding and Deleting Elements

The document object can also create and manipulate nodes.

Method	Description
`document.createElement(tag)`	Create a new element.
`document.createTextNode(text)`	Create a text node.
`parent.appendChild(node)`	Add a node as the last child.
`parent.insertBefore(new, ref)`	Insert before a reference node.
`parent.removeChild(node)`	Remove a node.
`parent.replaceChild(new, old)`	Replace an old node with a new one.
`document.write(text)`	Write directly to the page (not recommended).

Listing 4.5: Adding and removing nodes

```
<ul id="list">
  <li>Apple</li>
</ul>

<script>
  let ul = document.getElementById("list");

  // Create and append
  let li = document.createElement("li");
  li.textContent = "Banana";
  ul.appendChild(li);

  // Replace
  let newLi = document.createElement("li");
  newLi.textContent = "Mango";
  ul.replaceChild(newLi, li);

  // Remove
  newLi.remove();
</script>
```

4.2.4 Adding Event Handlers

The document object also allows attaching event handlers to elements.

Method	Description
`document.getElementById(id).onclick = function() {...}`	Assign a handler for onclick.
`element.addEventListener("event", handler)`	Attach modern event listeners.

Listing 4.6: Event handler example

```
<button id="btn">Click Me</button>

<script>
  let b = document.getElementById("btn");
  b.onclick = function() {
    alert("Clicked!");
  };
</script>
```

4.2.5 Common Document Properties and Collections

The document object exposes many properties for accessing parts of the page.

Property	Description	DOM Level
document.body	Returns the <body> element.	1
document.head	Returns the <head> element.	3
document.title	Gets/sets the <title> text.	1
document.URL	Returns the complete URL of the document.	1
document.baseURI	Absolute base URI of the document.	3
document.forms	Collection of all <form> elements.	1
document.images	Collection of all <img> elements.	1
document.links	Collection of all <a> and <area> with href.	1
document.scripts	Collection of all <script> elements.	3
document.cookie	Gets/sets cookies.	1
document.lastModified	Returns date/time of last update.	3
document.readyState	Returns loading status (loading, interactive, complete).	3
document.referrer	URL of referring document.	1

4.2.6 Examples Using Document Properties

Listing 4.7: Using document properties

```
<script>
  console.log(document.title);         // prints page title
  console.log(document.URL);           // prints current URL
  console.log(document.lastModified); // last updated time

  console.log(document.body);          // body element
  console.log(document.forms.length); // number of forms
  console.log(document.images.length);// number of images
</script>
```

4.2.7 Summary

- The document object is the entry point for all DOM operations.
- It provides methods to *find*, *create*, *update*, and *delete* elements.
- It exposes properties to quickly access body, head, forms, images, scripts, links, etc.
- Mastering the document object is the foundation of all DOM manipulation in JavaScript.

4.3. Introduction to Events

JavaScript is a programming language that runs inside the web browser. If we only write plain JavaScript without events, our programs would run once when the page is loaded and then stop. The page would look static and unresponsive.

To make web pages interactive, we need a way for the browser to *notify our program* whenever something important happens — for example, when the user clicks a button, presses a key, or submits a form. These notifications are called **events**.

In other words, an *event* is like a message sent from the browser to your JavaScript code saying:

> "Hey, something just happened, do you want to react to it?"

4.3.1 What is an Event?

An **event** is a signal that is fired (triggered) whenever a specific action occurs in the browser.

These actions can come from:

- **User actions:** clicking the mouse, pressing a key, typing into an input box, scrolling the page, or touching the screen on a mobile phone.
- **Browser actions:** finishing the loading of a page, resizing the browser window, or completing a network request.
- **Program actions:** JavaScript itself can create and dispatch a *custom event* to communicate between different parts of the program.

Every event carries with it some extra information, like:

- *Which element* it happened on (the `target`).
- *When* it happened (the `timeStamp`).
- *Details of the action* (for example: which mouse button was clicked, which key was pressed, or what text was typed).

This information is delivered through an **event object**, which your code can use to decide how to respond.

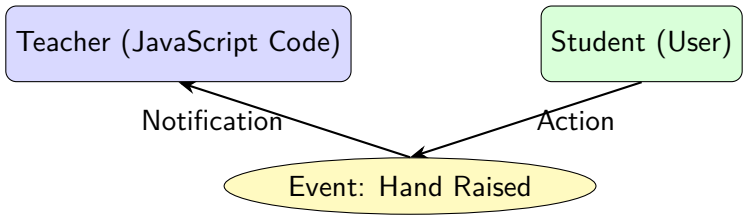
4.3.2 Real-World Analogy

Think of a classroom situation:

- The teacher says: "Raise your hand if you have a question."
- A student raises their hand. This action is an *event*.
- The teacher notices and responds: "Yes, what is your question?" This is like JavaScript running an *event handler* when the event occurs.

Without events, the teacher would have to continuously ask every student: "Do you have a question now? Do you have a question now?" — which would be inefficient.

Events provide a much smarter way: they automatically notify when something has happened.



This diagram shows how the **user action** (student raising hand) becomes an **event notification** to JavaScript (teacher), which then decides what to do.

4.3.3 Why are Events Important?

Events are essential because they allow JavaScript to:

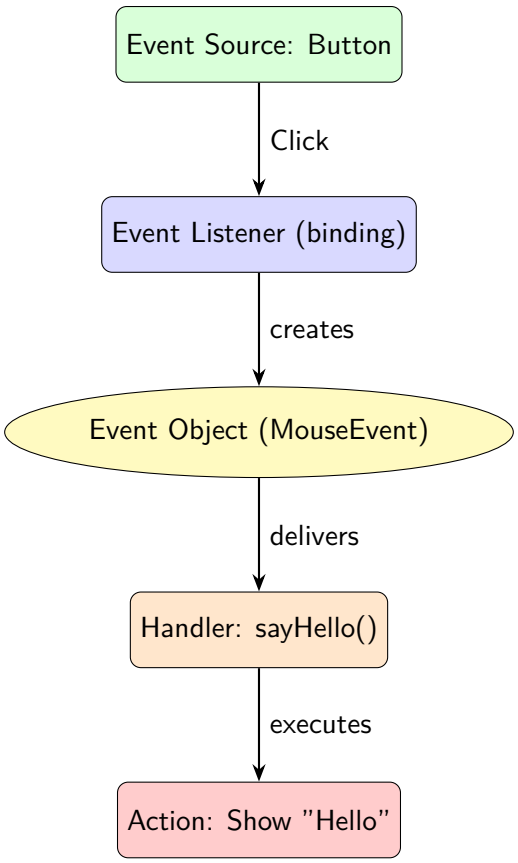
- **Make web pages interactive** — e.g., clicking a button to open a menu.
- **Validate data before submission** — e.g., checking a form field when the user clicks "Submit".
- **React to environment changes** — e.g., adjusting the layout when the window is resized.
- **Enable real-time communication** — e.g., dispatching custom events to notify different parts of the program.

4.3.4 Event Flow Example: Button Click

To understand how events work in practice, consider a simple button that says "Hello" when clicked. This example shows the relationship between the **event source**, the **event type**, the **listener (binding)**, the **event object**, and the **action (handler)**.

Listing 4.8: Button click with event listener

```
<button id="helloBtn">Click Me</button>

<script>
  // Step 1: Event source (the button element)
  const btn = document.getElementById('helloBtn');

  // Step 2: Bind event listener for type 'click'
  btn.addEventListener('click', sayHello);

  // Step 3: Define handler (action to perform)
  function sayHello(event) {
    // event is a MouseEvent object
    console.log("Event source:", event.target.id);
    alert("Hello from JavaScript!");
  }
</script>
```



Explanation of Flow:

1. The *event source* is the button element in the DOM.
2. When the user clicks, the browser fires a `click` event.
3. Because we bound a listener with `addEventListener`, the browser creates an **event object** (`MouseEvent`).
4. The event object is passed into our handler function (`sayHello`).
5. The handler runs the code inside it (here: showing an alert "Hello").

Summary: Events are the *bridge* between the user (or browser) and your JavaScript code. Without them, web pages would remain static documents. With them, we can build responsive, user-friendly, and dynamic applications.

4.4. JavaScript Events: Detailed Reference

Whenever an event is fired in the browser, it is represented as an **Event Object**. This object is automatically passed to event handlers and carries useful information such as:

- `type` – the event type (e.g., `click`, `keydown`).
- `target` – the element on which the event occurred.
- `currentTarget` – the element whose handler is currently running.
- `timeStamp` – when the event occurred.
- `bubbles` and `cancelable` – whether the event bubbles up and can be canceled.

- `preventDefault()` and `stopPropagation()` – methods to control event behavior.

Specialized event objects extend the base `Event` with additional properties (e.g., `MouseEvent`, `KeyboardEvent`, `FocusEvent`, etc.).

Table 4.6: Detailed list of JavaScript events, their event objects, and common purposes.

Event Name	Event Object Type	Purpose / Common Use-Cases
Mouse Events		
`click`	MouseEvent	Single click on an element (buttons, links, UI interactions).
`dblclick`	MouseEvent	Double-click, e.g. open item in file manager.
`mousedown`	MouseEvent	Press mouse button down (start drag operations).
`mouseup`	MouseEvent	Release mouse button (end drag or click).
`mousemove`	MouseEvent	Track pointer movement (drawing on canvas, hover effects).
`mouseover`	MouseEvent	Pointer enters element (hover menus).
`mouseout`	MouseEvent	Pointer leaves element.
`contextmenu`	MouseEvent	Right-click to open custom menus.
Keyboard Events		
`keydown`	KeyboardEvent	Key pressed down (detect shortcuts).
`keyup`	KeyboardEvent	Key released (finalize input, detect combos).
`keypress`	KeyboardEvent	(Legacy) Fired when a character key is pressed.
Form and Input Events		
`input`	InputEvent	Fired on every change while typing.
`change`	Event	Fired when input value committed (blur, selection change).
`submit`	SubmitEvent	Fired when form is submitted.
`reset`	Event	Fired when form reset button is pressed.
`focus`	FocusEvent	Element gains input focus.
`blur`	FocusEvent	Element loses input focus.
Window and Document Events		
`load`	Event	Entire page and resources finished loading.
`DOMContentLoaded`	Event	DOM tree loaded (before images/styles).
`beforeunload`	Event	Triggered before page unload, can show confirmation.
`unload`	Event	Page being unloaded (cleanup).
`resize`	UIEvent	Browser window resized (responsive design).
`scroll`	UIEvent	User scrolled viewport (lazy loading, infinite scroll).
Pointer Events (Unified Mouse/Touch/Pen)		
`pointerdown`	PointerEvent	Pointer contact starts.
`pointerup`	PointerEvent	Pointer lifted.
`pointermove`	PointerEvent	Pointer moved (track stylus).
`pointerenter`	PointerEvent	Pointer enters element (like mouseover).
`pointerleave`	PointerEvent	Pointer leaves element.
`gotpointercapture`	PointerEvent	Element gains exclusive pointer capture.
`lostpointercapture`	PointerEvent	Element loses pointer capture.

Continued on next page

Table 4.6 – continued from previous page

Event Name	Event Object Type	Purpose / Common Use-Cases
Touch Events (Mobile)		
`touchstart`	TouchEvent	Finger touches screen.
`touchmove`	TouchEvent	Finger moves on screen (swipe, drag).
`touchend`	TouchEvent	Finger lifted.
`touchcancel`	TouchEvent	System cancels touch (gesture interrupt).
Drag and Drop Events		
`dragstart`	DragEvent	Dragging started.
`drag`	DragEvent	Element being dragged.
`dragenter`	DragEvent	Drag enters droppable area.
`dragover`	DragEvent	Dragging over target (must `preventDefault` to allow drop).
`dragleave`	DragEvent	Leaving drop target.
`drop`	DragEvent	Dropped on target.
`dragend`	DragEvent	Drag operation ended.
Clipboard Events		
`copy`	ClipboardEvent	User copies content.
`cut`	ClipboardEvent	User cuts content.
`paste`	ClipboardEvent	User pastes content.
Media Events (Audio/Video)		
`play`	Event	Playback started.
`pause`	Event	Playback paused.
`ended`	Event	Media finished.
`timeupdate`	Event	Playback position updated.
`volumechange`	Event	Volume or mute state changed.
`error`	Event	Media error.
Animation and Transition Events		
`animationstart`	AnimationEvent	CSS animation started.
`animationend`	AnimationEvent	Animation ended.
`animationiteration`	AnimationEvent	Animation repeated.
`transitionstart`	TransitionEvent	CSS transition started.
`transitionend`	TransitionEvent	Transition ended.
Custom Events		
`CustomEvent`	CustomEvent	Developer-defined events with `detail` payloads. Used for decoupling components.
Network / Realtime Events		
`message`	MessageEvent	WebSocket or postMessage communication.
`open`	Event	Connection opened.
`close`	CloseEvent	Connection closed.
`error`	Event	Network or protocol error.

4.4.1 Notes on commonly used event objects ›

MouseEvent. The most common properties are the coordinates (`clientX`, `clientY` relative to the

viewport; `pageX`, `pageY` relative to the document) and `button` to determine which mouse button triggered the event. Modifier keys can be read via `ctrlKey`, `shiftKey`, `altKey`, `metaKey`. Use `preventDefault()` to prevent the browser's default handling (for example, preventing a link from following its URL).

KeyboardEvent. Prefer `keydown` and `keyup` for reliable behavior. `key` contains the semantic value (e.g. `'Enter'`, `'a'`), while `code` represents the physical key on the keyboard (e.g. `'KeyA'`). `repeat` indicates whether the key event is a repeating key when held down.

InputEvent. Fired on editable elements when the value changes — gives access to the latest input and often has a `data` property describing the inserted characters. This event is useful for live validation and instant search.

PointerEvent & TouchEvent. On modern browsers prefer `PointerEvent` for unified pointer handling (mouse/touch/pen). For legacy or fine-grained multi-touch control, `TouchEvent` provides lists of touch points.

CustomEvent. Use `new CustomEvent(name, { detail: yourObject, bubbles: true/false })` to send structured data through the DOM. Handlers read data via `event.detail`.

4.4.2 Practical Examples (widely used events) ›

Each event in JavaScript delivers an *event object* that provides useful data to the handler. Below are practical examples organized by event object type. Each example is a complete, minimal snippet with an explanation of its purpose.

Listing 4.9: Button click example

1. Mouse Events (MouseEvent)

```
<button id="clickBtn">Click me</button>

<script>
  const btn = document.getElementById('clickBtn');
  btn.addEventListener('click', function (event) {
    // event is a MouseEvent
    console.log('Clicked at', event.clientX, event.clientY);
    alert('Button clicked!');
  });
</script>
```

Use-case: Basic UI triggers (open dialogs, toggles, menus).

Listing 4.10: Mouseover / mouseout example

```
<div id="box" style="width:100px;height:100px;background:lightblue;"></
   div>

<script>
  const box = document.getElementById('box');
```

```
  box.addEventListener('mouseover', e => {
    box.style.background = 'orange';
  });
  box.addEventListener('mouseout', e => {
    box.style.background = 'lightblue';
  });
</script>
```

Use-case: Hover effects, tooltips, interactive menus.

Listing 4.11: Keyboard shortcut (Ctrl+S) example

2. Keyboard Events (KeyboardEvent)

```
<script>
  window.addEventListener('keydown', function (e) {
    if ((e.ctrlKey || e.metaKey) && e.key === 's') {
      e.preventDefault(); // stop Save dialog
      console.log('Save shortcut triggered');
    }
  });
</script>
```

Use-case: Implementing keyboard shortcuts.

Listing 4.12: Detecting key release (keyup)

```
<input id="name" placeholder="Type your name">

<script>
  const nameInput = document.getElementById('name');
  nameInput.addEventListener('keyup', function (e) {
    console.log('Last key released:', e.key);
  });
</script>
```

Use-case: Validation after typing, auto-complete triggers.

Listing 4.13: Live validation using input event

3. Form / Input Events (InputEvent, SubmitEvent)

```
<input id="email" placeholder="Enter email">
<p id="msg"></p>

<script>
  const email = document.getElementById('email');
  const msg = document.getElementById('msg');
```

```
  email.addEventListener('input', function (e) {
    if (e.target.value.includes('@')) {
      msg.textContent = 'Looks like an email.';
    } else {
      msg.textContent = 'Invalid email format.';
    }
  });
</script>
```

Use-case: Live form validation.

Listing 4.14: Preventing form submission for validation

```
<form id="loginForm">
  <input name="username" required>
  <button type="submit">Login</button>
</form>

<script>
  const form = document.getElementById('loginForm');
  form.addEventListener('submit', function (e) {
    if (!form.username.value.trim()) {
      e.preventDefault();
      alert('Username required!');
    }
  });
</script>
```

Use-case: Custom form validation, AJAX submission.

Listing 4.15: DOMContentLoaded example

4. Window / Document Events (UIEvent, Event)

```
<script>
  document.addEventListener('DOMContentLoaded', function () {
    console.log('DOM ready - safe to query elements');
  });
</script>
```

Use-case: Run initialization code after HTML parsing.

Listing 4.16: Window resize event

```
<script>
  window.addEventListener('resize', function () {
    console.log('Resized:', window.innerWidth, window.innerHeight);
```

```
  });
</script>
```

Use-case: Responsive layout adjustments.

Listing 4.17: Pointer down/up tracking

5. Pointer / Touch Events (PointerEvent, TouchEvent)

```
<div id="draw" style="width:200px;height:150px;border:1px solid #aaa"><
   /div>

<script>
  const draw = document.getElementById('draw');
  draw.addEventListener('pointerdown', e => {
    console.log('Pointer down', e.pointerId, e.pointerType);
    draw.setPointerCapture(e.pointerId);
  });
  draw.addEventListener('pointerup', e => {
    draw.releasePointerCapture(e.pointerId);
    console.log('Pointer up');
  });
</script>
```

Use-case: Drawing, dragging, gesture detection.

Listing 4.18: Basic touch example (mobile)

```
<div id="touchZone" style="width:200px;height:80px;border:1px dashed
   gray;">
  Touch inside here
</div>

<script>
  const zone = document.getElementById('touchZone');
  zone.addEventListener('touchstart', e => {
    console.log('Touches:', e.touches.length);
  });
</script>
```

Use-case: Mobile gestures, swipe detection.

Listing 4.19: File drop example

6. Drag and Drop (DragEvent)

```
<div id="dropzone" style="width:250px;height:100px;border:2px dashed
   #666">
```

```
  Drop files here
</div>

<script>
  const dz = document.getElementById('dropzone');
  dz.addEventListener('dragover', e => e.preventDefault());
  dz.addEventListener('drop', e => {
    e.preventDefault();
    console.log('Dropped:', e.dataTransfer.files);
  });
</script>
```

Use-case: File uploads via drag and drop.

Listing 4.20: Custom paste handling

7. Clipboard Events (ClipboardEvent)

```
<input id="pasteBox" placeholder="Paste text here">

<script>
  const box = document.getElementById('pasteBox');
  box.addEventListener('paste', function (e) {
    e.preventDefault();
    const text = e.clipboardData.getData('text');
    console.log('Pasted text:', text);
  });
</script>
```

Use-case: Sanitize pasted input, custom formatting.

Listing 4.21: Play/pause event listeners

8. Media Events (MediaElement Event)

```
<video id="vid" src="movie.mp4" width="320" controls></video>

<script>
  const vid = document.getElementById('vid');
  vid.addEventListener('play', () => console.log('Playing...'));
  vid.addEventListener('pause', () => console.log('Paused.'));
</script>
```

Use-case: Sync UI with playback state.

Listing 4.22: CSS animation events

9. Animation / Transition Events (AnimationEvent)

```
<div id="box" style="width:50px;height:50px;background:red;
                     animation: move 2s linear infinite;"></div>

<style>
@keyframes move {
  from { transform: translateX(0); }
  to   { transform: translateX(200px); }
}
</style>

<script>
  const box = document.getElementById('box');
  box.addEventListener('animationiteration', e => {
    console.log('Animation loop finished', e.elapsedTime);
  });
</script>
```

Use-case: Trigger logic when animations complete.

Listing 4.23: Custom event communication

10. Custom Events (CustomEvent)

```
<script>
  // Component A: dispatch
  const data = { id: 100, qty: 2 };
  const ev = new CustomEvent('cart:add', { detail: data });
  document.dispatchEvent(ev);

  // Component B: listen
  document.addEventListener('cart:add', e => {
    console.log('Cart updated:', e.detail);
  });
</script>
```

Use-case: Decoupled communication between components.

4.4.3 Tips for student exercises

- students to inspect the event object in DevTools: `console.log(event)` and expand it to explore properties.
- Show the difference between `event.target` and `event.currentTarget`.
- exercise: detect mouse button, detect modifier keys, implement keyboard shortcuts, create a small drag-and-drop interface for arranging items.
- Emphasize performance: avoid heavy work inside `scroll` or `mousemove` handlers — debounce or throttle when necessary.

- When targeting mobile devices, find the difference between `touch` and `pointer` events and recommend `PointerEvent` for modern apps.

4.5. Binding Events to Elements

When we want our JavaScript code to respond to user or browser actions, we must *bind* (attach) an event handler to an element. An **event handler** is simply a function that will be executed whenever the specified event occurs.

There are three main approaches to binding events in JavaScript:

1. Inline HTML attributes (old style, rarely recommended today).
2. DOM property assignment (often called DOM0 binding).
3. `addEventListener` (modern, flexible, recommended).

Each method has advantages and disadvantages, which we will explore with examples.

4.5.1 1. Inline HTML Attribute Binding

This was the earliest method of attaching events in HTML. The event name (like `onclick`) is written directly as an attribute in the element's HTML tag, and the value is JavaScript code to execute.

Listing 4.24: Inline event binding example

```
<!DOCTYPE html>
<html>
  <body>
    <!-- Inline attribute binding -->
    <button onclick="sayHello()">Click Me</button>

    <script>
      function sayHello() {
        alert("Hello from inline binding!");
      }
    </script>
  </body>
</html>
```

Advantages:

- Very simple to write.
- Useful for quick tests or demos.

Disadvantages:

- Mixes HTML and JavaScript, making code harder to maintain.
- Hard to attach multiple handlers to the same event.
- Not considered best practice for modern applications.

4.5.2 2. DOM Property Binding (`element.onclick`)

Here, we select the element in JavaScript and assign a function directly to its event property (such as `onclick`, `onmouseover`, etc.). This is often called DOM0 binding.

Listing 4.25: DOM property onclick example

```
<!DOCTYPE html>
<html>
  <body>
    <button id="myBtn">Click me</button>

    <script>
      const btn = document.getElementById("myBtn");

      // Assign a handler function to the onclick property
      btn.onclick = function () {
        alert("Button clicked using DOM property!");
      };

      // Reassigning replaces the old handler
      btn.onclick = function () {
        console.log("New handler replaced the old one.");
      };
    </script>
  </body>
</html>
```

Key point: Only *one* handler can exist per event property. Reassigning overwrites the previous one.

4.5.3 3. Modern Binding: `addEventListener`

The recommended way is to use `addEventListener`. This allows attaching multiple handlers for the same event, fine control over event propagation (capture/bubble), and removal of specific listeners.

Listing 4.26: addEventListener example

```
<!DOCTYPE html>
<html>
  <body>
    <button id="btn">Click Me</button>

    <script>
      const btn = document.getElementById("btn");

      // Attach a click event listener
```

```
      function handler1() {
        alert("Handler 1 executed");
      }

      function handler2() {
        console.log("Handler 2 executed");
      }

      btn.addEventListener("click", handler1);
      btn.addEventListener("click", handler2);

      // Remove one handler later
      setTimeout(() => {
        btn.removeEventListener("click", handler1);
        console.log("Handler 1 removed");
      }, 5000);
    </script>
  </body>
</html>
```

4.5.4 Comparison of Binding Methods

JavaScript provides three main approaches to bind events to elements:

- **Inline attributes (HTML event attributes)** Writing the event handler directly in the HTML element using attributes like `onclick`, `onchange`, etc. **Pros:** Very quick for demos or small scripts. **Cons:** Mixes HTML and JavaScript, makes code hard to maintain, and allows only one handler inline.

```
<!-- Inline event attribute -->
<button onclick="alert('Hello!')">Click Me</button>
```

- **DOM property (DOM0)** Assigning a handler to the event property of the DOM object, such as `element.onclick = function() {...}`. **Pros:** Cleaner than inline, keeps HTML and JavaScript separate. **Cons:** Only one handler can be assigned — assigning another will overwrite the previous one.

```
<!-- DOM0 event binding -->
<button id="btn1">Click</button>
<script>
  const btn = document.getElementById("btn1");
  btn.onclick = function() {
    alert("First handler");
  };
  // Overwrites the previous handler
  btn.onclick = function() {
    alert("Second handler");
  };
```

```
</script>
```

- **`addEventListener` (DOM2)** The modern and recommended way: `element.addEventListener(event, handler, useCapture)`. **Pros:**
  - Multiple handlers can be attached to the same event.
  - Supports capturing vs. bubbling phases.
  - Handlers can be easily removed with `removeEventListener`.

  Cons: Slightly more verbose.

```
<!-- DOM2 event binding -->
<button id="btn2">Click</button>
<script>
  const btn2 = document.getElementById("btn2");
  function handler1() { console.log("Handler 1"); }
  function handler2() { console.log("Handler 2"); }

  btn2.addEventListener("click", handler1);
  btn2.addEventListener("click", handler2);

  // Later, remove one handler
  btn2.removeEventListener("click", handler1);
</script>
```

Comparison Table:

Method	Syntax Example	Advantages	Disadvantages
Inline attributes	`<button onclick="fn()">Click</button>`	Very quick, visible in HTML, no extra JS file required	Poor separation of concerns, hard to maintain, only one handler, outdated.
DOM property (DOM0)	`element.onclick = fn;`	Simple syntax, JS kept separate from HTML	Only one handler per event property, overwrites previous handler.
addEventListener (DOM2)	`element.addEventListener('click', fn);`	Multiple handlers, can remove handlers, supports capture/bubble, modern best practice	Slightly longer syntax, may not work in very old browsers (pre-IE9).

Conclusion: For modern applications, always prefer `addEventListener` because of its flexibility, maintainability, and compatibility with event propagation models.

4.5.5 Practical Example Comparing All Three

The following example shows the same button with all three binding styles for demonstration. In practice, only use one style consistently (preferably `addEventListener`).

Listing 4.27: Comparison of binding methods

```
<!DOCTYPE html>
<html>
  <body>
    <!-- Inline attribute -->
    <button id="btn1" onclick="alert('Inline binding clicked!')">
      Inline Button
    </button>

    <button id="btn2">DOM Property Button</button>
    <button id="btn3">addEventListener Button</button>

    <script>
      // DOM property
      document.getElementById("btn2").onclick = function () {
        alert("DOM property binding clicked!");
      };

      // addEventListener
      const btn3 = document.getElementById("btn3");
      btn3.addEventListener("click", function () {
        alert("addEventListener binding clicked!");
      });
    </script>
  </body>
</html>
```

4.6. Event Delegation

In JavaScript, every event has a **source**, is wrapped into an **event object**, and is handled by an **event listener**. Event Delegation is a powerful technique where instead of attaching separate listeners to every child element, we attach *a single listener* to a common parent element and determine which child triggered the event. This improves efficiency and also works for dynamically added elements.

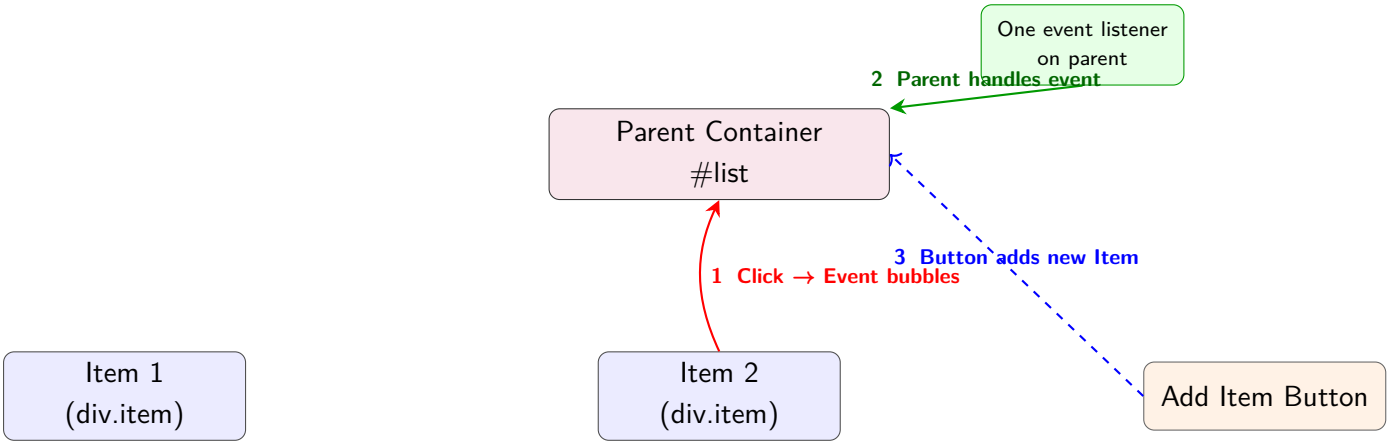
4.6.1 Key Concepts

- **Event Source:** The element where the event originates. Example: when a user clicks on `Item 3`, that `<div class="item">` is the event source.
- **Event:** The signal created when the action happens. The browser generates an `Event` (or a more specific subclass like `MouseEvent`) with details.
- **Event Listener:** A function attached to an element that runs when a specific event type occurs. Example: `list.addEventListener('click', handler)`.
- **Event Delegation:** Attaching the listener to a parent (such as the `#list` container) instead of to

every `.item`. The event bubbles up from the source to the parent, and the listener checks which child triggered it using `event.target` or `closest()`.

4.6.2 Event Flow Diagram

The following diagram illustrates event delegation for a list of items inside a parent container. A click on `Item 2` bubbles up to the `list` parent, where a single listener handles all items.



4.6.3 Event Delegation Example Code

Listing 4.28: Event delegation for a dynamic list

```
<div id="list">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
</div>
<button id="addItem">Add Item</button>

<script>
  const list = document.getElementById('list');
  const addBtn = document.getElementById('addItem');
  let count = 3;

  // delegation: one listener on parent
  list.addEventListener('click', function (e) {
    const item = e.target.closest('.item');
    if (!item) return;
    alert('Clicked: ' + item.textContent);
  });

  // dynamically add new items
  addBtn.addEventListener('click', () => {
```

```
    count++;
    const el = document.createElement('div');
    el.className = 'item';
    el.textContent = 'Item ' + count;
    list.appendChild(el);
  });
</script>
```

4.6.4 Step-by-Step Explanation

Let us carefully walk through how the event delegation example works:

1. The HTML starts with three `.item` elements inside a container `<div id="list">`. These represent clickable items in the list.
2. A button (`Add Item`) allows the user to dynamically insert new items into the list. This demonstrates that delegation works even with content added at runtime.
3. Instead of attaching a separate `click` listener to each `.item`, a single `click` listener is attached to the parent container `#list`. This is the essence of delegation.
4. When a child `.item` is clicked, the event originates from that element (this is the *event source*). The browser then lets the event bubble upwards through the DOM tree until it reaches the parent `#list`.
5. Inside the event handler, `event.target.closest('.item')` is used to check which element triggered the click. This ensures we only respond if the click came from an item, not from blank space in the container.
6. Because the listener is on the parent, all newly created items (added via the button) also work automatically with the same event handler. There is no need to re-bind listeners for new elements.

4.6.5 Why Event Delegation is Useful

Event delegation offers multiple benefits:

- **Performance:** Instead of attaching hundreds of listeners to each child element (which consumes memory and slows down the page), only a single listener is attached to the parent. This is more efficient.
- **Dynamic content:** When new elements are added dynamically to the DOM (for example, when adding new list items), they automatically become interactive without writing extra code.
- **Cleaner code:** The logic is centralized in one handler instead of being scattered across multiple listeners, making the code easier to maintain and debug.

4.7. Replace/Update Child Example

Sometimes we need to **replace an existing element** in the DOM with a new one. JavaScript provides the `replaceChild()` method for this purpose.

Listing 4.29: Replace child using replaceChild

```
<div id="container">
  <p id="old">Old text</p>
</div>
<button id="update">Update</button>

<script>
  const container = document.getElementById('container');
  const old = document.getElementById('old');
  document.getElementById('update').onclick = () => {
    const replacement = document.createElement('p');
    replacement.textContent = 'New updated content';
    container.replaceChild(replacement, old);
  };
</script>
```

In this example:

- The container initially has one child paragraph (<p> with id `old`).
- Clicking the `Update` button creates a new <p> element.
- The `replaceChild()` method replaces the old child with the new one.

This is particularly useful when you want to update the structure of the DOM (not just the text or attributes).

4.8. Event Propagation & Listener Options

When an event occurs, it passes through three distinct phases:

1. **Capturing phase:** The event travels from the window down through the DOM tree to the target.
2. **Target phase:** The event reaches the target element.
3. **Bubbling phase:** The event bubbles back up from the target to the window.

By default, most event listeners run in the bubbling phase. However, we can control behavior with options:

- `{capture:true}` – Listen during the capturing phase instead of bubbling.
- `{once:true}` – The listener runs only once, then is removed automatically.
- `{passive:true}` – Hints to the browser that the handler will not call `preventDefault()`. This improves performance for scroll/touch events.

Common event object methods:

- `preventDefault()` – Stops the browser's default action (e.g., stopping a link from navigating).
- `stopPropagation()` – Prevents the event from continuing to bubble up the DOM.

4.9. Custom Events

Sometimes you want parts of your application to communicate by sending their own events. This is done with the `CustomEvent` constructor.

Listing 4.30: Create & dispatch a custom event

```
const ev = new CustomEvent('cart:added', {
  detail: { id: 42, qty: 1 },
  bubbles: true
});

document.querySelector('body').dispatchEvent(ev);

document.body.addEventListener('cart:added', (e) => {
  console.log('Added to cart', e.detail);
});
```

Explanation:

- The event is created with a name (`'cart:added'`).
- The `detail` property carries extra data (here, product ID and quantity).
- The event is dispatched on the `body`, and any listener on `body` or its ancestors can catch it.
- The handler receives the event object and accesses the custom data via `event.detail`.

4.10. Common Pitfalls

- **Anonymous handlers cannot be removed.** Always use named functions if you plan to call `removeEventListener`
- **Memory leaks.** Leaving many unused listeners attached can waste memory.
- **Passive listeners and preventDefault.** If a listener is marked `passive:true`, calling `preventDefault()` will be ignored.
- **Incorrect target handling.** Always check `event.target` or use `closest()` to avoid responding to clicks on nested elements unintentionally.

4.11. Exercises

Try the following tasks to reinforce your understanding:

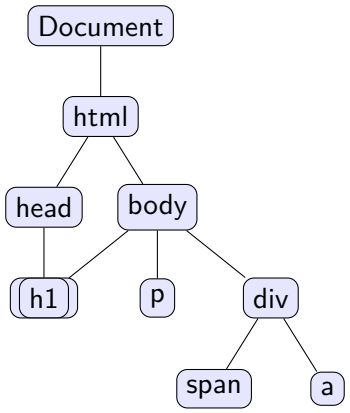
1. Create a button that shows an alert message when clicked. (*Practice basic click events.*)
2. Add a `keydown` listener on the window that logs which key was pressed. (*Practice keyboard event handling.*)
3. Build a form with one input field. Prevent the form from submitting if the field is empty. (*Practice `preventDefault()` and validation.*)
4. Make a dynamic list with items. Use event delegation to handle clicks on all items, including new ones added dynamically. (*Practice delegation and bubbling.*)

5. Create and dispatch a custom event named `'user:login'` that passes user details in `detail`, and handle it elsewhere in the code. (*Practice CustomEvent and communication.*)

4.12. UNIT II – DOM & JavaScript Events - Questions and Answers

4.12.1 What is the Document Object Model (DOM)? Explain its tree struct

- DOM is a programming interface for HTML and XML documents.
- It represents the page as a tree structure of nodes (elements, attributes, text).
- JavaScript can manipulate the DOM to change structure, style, and content dynamically.



4.12.2 Write JavaScript code to select elements using `getElementById()`, g

- `getElementById()` – selects element by ID.
- `getElementsByClassName()` – returns a collection of elements.
- `querySelector()` – selects first matching element using CSS selector.

```
document.getElementById("header").style.color = "blue";
let items = document.getElementsByClassName("list-item");
items[0].style.fontWeight = "bold";
let para = document.querySelector("p.intro");
para.style.background = "yellow";
```

4.12.3 Explain DOM manipulation methods with examples for creating, upd

- **createElement()** – creates a new element.
- **appendChild()** – adds new element to parent.
- **textContent / innerHTML** – updates content.
- **removeChild()** – deletes element.

```
let div = document.createElement("div");
div.textContent = "Hello DOM";
document.body.appendChild(div);

div.textContent = "Updated Content";

document.body.removeChild(div);
```

4.12.4 Write JavaScript code to change the content and attributes of HTM

```
document.getElementById("title").textContent = "New Title";
document.getElementById("myImg").setAttribute("src", "newpic.jpg");
```

4.12.5 What is an event in JavaScript? Explain different types of browser e

- Events are actions or occurrences in the browser (e.g., click, keypress).
- Types:
  - **Mouse Events** – click, dblclick, mouseover.
  - **Keyboard Events** – keydown, keyup.
  - **Form Events** – submit, change, focus.
  - **Window Events** – load, resize, scroll.

```
document.getElementById("btn").onclick = function() {
  alert("Button clicked!");
};
```

4.12.6 Write a program to demonstrate event binding using addEventListe

```
let btn = document.getElementById("btn");
btn.addEventListener("click", function() {
  alert("Hello from addEventListener!");
});
```

4.12.7 Explain event delegation with a suitable JavaScript program.

- Event delegation allows handling events for multiple child elements using a single parent.

- Improves performance by reducing multiple listeners.

```
document.getElementById("list").addEventListener("click", function(e) {
  if (e.target.tagName === "LI") {
    console.log("You clicked:", e.target.textContent);
  }
});
```

4.12.8 Compare inline event handling, DOM property binding, and addEvent

- **Inline Handling**

```
<button onclick="alert('Hi')">Click</button>
```

- **DOM Property Binding**

```
document.getElementById("btn").onclick = function() { alert("Hello
    "); };
```

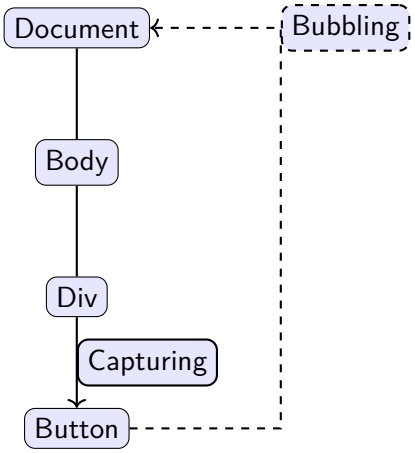
- **addEventListener()**

```
document.getElementById("btn").addEventListener("click", () =>
    alert("Hi"));
```

4.12.9 What is event bubbling and event capturing? Explain with examples

- **Event Bubbling** – event propagates from target element upward to parent nodes.
- **Event Capturing** – event propagates from root down to target element.

```
parent.addEventListener("click", () => console.log("Parent"), true); //
    capturing
child.addEventListener("click", () => console.log("Child")); //
   bubbling
```



4.12.10 Write a JavaScript program to display the name of a list item when

```
document.getElementById("menu").addEventListener("click", function(e) {
  if (e.target.tagName === "LI") {
    alert("Item: " + e.target.textContent);
  }
});
```

CHAPTER 5

5 UNIT III – Forms & MERN

5.1. Form Enhancement and Validation

Forms are the backbone of web applications as they allow user input and submission of data. HTML5 provides various tags and attributes to make forms more interactive, accessible, and secure.

5.1.1 Form Element and Attributes

The `<form>` element defines a form that can contain input fields, checkboxes, radio buttons, and other interactive controls.

- **action** – URL where form data is sent after submission.
- **method** – HTTP method used (`GET` or `POST`).
- **enctype** – encoding type for submitted data (`application/x-www-form-urlencoded`, `multipart/form-data`, etc.).
- **name** – identifier for the form.

```
<form action="/submit" method="post" enctype="multipart/form-data" name
    ="userForm">
  <!-- form controls here -->
</form>
```

5.1.2 Input Types and Attributes

The `<input>` tag is the most common form control. HTML5 provides various types and attributes.

- **Types**: text, password, email, number, checkbox, radio, file, date, color, range, url, etc.
- **Attributes**:
  - `required` – field must be filled.
  - `disabled` – field is not editable and excluded from submission.
  - `readonly` – field is displayed but not editable (submitted).
  - `pattern` – regex pattern for validation.
  - `min`, `max` – range for numeric/date inputs.
  - `maxlength` – maximum number of characters.
  - `placeholder` – hint text.

```
<input type="text" name="username" required placeholder="Enter name" />
<input type="email" name="email" required />
<input type="password" name="pwd" minlength="6" />
<input type="number" min="1" max="100" />
<input type="file" name="resume" accept=".pdf" />
```

The <input> element supports many types. Each type defines how the field behaves and what kind of data it accepts. Table 5.1 shows the complete list of input types in HTML5 with description and examples.

Input Type	Description	Example
text	Single-line text field	`<input type="text" name="username">`
password	Password field (characters hidden)	`<input type="password" name="pwd">`
email	Must be valid email	`<input type="email" name="email">`
number	Numeric input with min/max	`<input type="number" min="1" max="100">`
url	Must be valid URL	`<input type="url" name="website">`

Input Type	Description	Example
tel	Telephone number (no validation by default)	`<input type="tel" name="phone">`
search	Search field with styling	`<input type="search" name="query">`
color	Color picker	`<input type="color" name="favcolor">`
date	Date picker (YYYY-MM-DD)	`<input type="date" name="dob">`
datetime-local	Date and time (no timezone)	`<input type="datetime-local" name="appt">`
month	Select month and year	`<input type="month" name="billMonth">`
week	Select week of year	`<input type="week" name="weekNum">`

Input Type	Description	Example
time	Select time (HH:MM)	`<input type="time" name="meeting">`
checkbox	Multiple selections allowed	`<input type="checkbox" name="hobby" value="sports"> Sports`
radio	Single selection from a group	`<input type="radio" name="gender" value="male"> Male`
file	File upload	`<input type="file" name="resume" accept=".pdf">`
range	Slider control	`<input type="range" min="0" max="100" step="10">`
hidden	Hidden field	`<input type="hidden" name="userid" value="123">`

Input Type	Description	Example
image	Submit button with image	`<input type="image" src="submit.png" width="100">`
reset	Reset form fields	`<input type="reset" value="Clear">`
submit	Submit the form	`<input type="submit" value="Register">`
button	Generic button (needs JS)	`<input type="button" value="Click" onclick="alert('Hi')">`

Table 5.1: HTML5 Input Types with Description and Examples

5.1.3 Other Form Controls

HTML provides additional elements for better input handling.

- **<select>, <option>, <optgroup>** – dropdown menus with grouped choices.
- **<textarea>** – multi-line text input.
- **<datalist>** – input suggestions with predefined values.
- **<fieldset>, <legend>** – group form fields and label them.

```
<select name="course">
  <optgroup label="Programming">
    <option value="js">JavaScript</option>
    <option value="py">Python</option>
  </optgroup>
  <optgroup label="Databases">
    <option value="sql">SQL</option>
```

```
    <option value="mongo">MongoDB</option>
  </optgroup>
</select>

<textarea name="comments" rows="4" cols="30"></textarea>

<input list="browsers" name="browser" />
<datalist id="browsers">
  <option value="Chrome">
  <option value="Firefox">
  <option value="Edge">
</datalist>

<fieldset>
  <legend>User Information</legend>
  <input type="text" name="fname" placeholder="First Name" />
  <input type="text" name="lname" placeholder="Last Name" />
</fieldset>
```

5.1.4 Form Validation

Validation ensures that form data meets expected rules before submission. Good validation has two layers:

- **Client-side (HTML5 + JavaScript)**: immediate feedback to the user; improves UX and reduces unnecessary requests to server.
  Note: Never rely solely on client-side validation for security.
- **Server-side**: final authoritative validation performed on the server to ensure data integrity and security.

Key validation techniques:

- Use HTML5 attributes: `required`, `type` (email, number, url), `pattern`, `min`, `max`, `minlength`, `maxlength`, `accept` for files.
- Use JavaScript for complex checks (password strength, cross-field checks like password/confirm, file size/type checks, aggregate rules).
- Use `setCustomValidity()` to supply custom browser error messages that integrate with HTML5 validity UI.
- Prevent form submission with `event.preventDefault()` when validation fails.

5.1.4.1 *Comprehensive HTML5 built-in validation example (many input types + pattern attributes)*

This example demonstrates many input types and uses built-in validation attributes including `pattern`, `required`, `min`, `max`, `accept`, and `minlength`/`maxlength`. The browser will show default validation UI when constraints fail.

```
<form id="fullForm" action="/submit" method="post" enctype="multipart/
    form-data">
```

```
<!-- Text and name -->
<label>Full Name: <input type="text" name="fullname" required
   minlength="3" maxlength="100"></label>

<!-- Email with built-in checking -->
<label>Email: <input type="email" name="email" required></label>

<!-- Telephone with pattern (example pattern allows digits, space, +
   and -) -->
<label>Phone:
  <input type="tel" name="phone" pattern="[\+0-9\-\s]{7,20}"
     placeholder="+91 99999-99999" required>
</label>

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

<!-- Password with pattern (at least 1 uppercase, 1 lowercase, 1
   digit, 1 special, min 8) -->
<label>Password:
  <input type="password" name="pwd"
         pattern="(?=^.{8,}$)(?=.*\d)(?=.*[A-Z])(?=.*[a-z])(?=.*\W).*
            $"
         title="Min 8 chars, with uppercase, lowercase, digit and
            special char" required>
</label>

<!-- Number with min and max -->
<label>Age: <input type="number" name="age" min="18" max="120"
   required></label>

<!-- Date/time inputs -->
<label>Date of Birth: <input type="date" name="dob" required></label>
<label>Appointment: <input type="datetime-local" name="appt"></label>

<!-- Range slider -->
<label>Rating: <input type="range" name="rating" min="0" max="10"
   step="1"></label>

<!-- Checkbox group (require at least one using JS or use required on
    first checkbox in modern browsers) -->
<fieldset>
  <legend>Hobbies (choose at least one)</legend>
  <label><input type="checkbox" name="hobby" value="sports"> Sports</
     label>
```

```
    <label><input type="checkbox" name="hobby" value="music"> Music</
        label>
    <label><input type="checkbox" name="hobby" value="reading"> Reading
        </label>
  </fieldset>

  <!-- Radio buttons -->
  <fieldset>
    <legend>Gender</legend>
    <label><input type="radio" name="gender" value="male" required>
        Male</label>
    <label><input type="radio" name="gender" value="female"> Female</
        label>
    <label><input type="radio" name="gender" value="other"> Other</
        label>
  </fieldset>

  <!-- Select, optgroup -->
  <label>Course:
    <select name="course" required>
      <optgroup label="Web">
        <option value="js">JavaScript</option>
        <option value="react">React</option>
      </optgroup>
      <optgroup label="Data">
        <option value="sql">SQL</option>
        <option value="mongo">MongoDB</option>
      </optgroup>
    </select>
  </label>

  <!-- Textarea -->
  <label>Comments:
    <textarea name="comments" rows="4" cols="40" maxlength="500"></
        textarea>
  </label>

  <!-- Datalist -->
  <label>Browser:
    <input list="browsers" name="browser" placeholder="Choose browser">
    <datalist id="browsers">
      <option value="Chrome"><option value="Firefox"><option value="
          Edge"><option value="Safari">
    </datalist>
  </label>
```

```
  <!-- File input with accept (only pdf or image) -->
  <label>Resume:
    <input type="file" name="resume" accept=".pdf,application/pdf,image
       /*" required>
  </label>

  <!-- Hidden, readonly, disabled demonstration -->
  <input type="hidden" name="userid" value="12345">
  <label>Read-only ref: <input type="text" name="ref" value="REF-2025"
     readonly></label>
  <label>Disabled demo: <input type="text" value="Disabled" disabled></
     label>

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

5.1.4.2 *Theory: how patterns work and common regex examples*

- `pattern` attribute uses a regular expression (without surrounding slashes). Browser tests the entire field value against this regex.
- Use `title` to explain the required format to users (shown by many browsers on validation failure).
- Common example patterns:
  - Phone: `[\+0-9\-\s]{7,20}` — allows +, digits, spaces, hyphens, length 7–20.
  - Strong password: `(?=^.{8,}$)(?=.*\d)(?=.*[A-Z])(?=.*[a-z])(?=.*\W).*$` — enforces min length + different character classes.
  - Simple username: `[A-Za-z0-9_]{3,20}` — letters, digits, underscore; 3–20 chars.
  - ISO date is handled by `type="date"`; avoid complex date regex.

5.1.4.3 *Custom (onsubmit) validation using JavaScript*

The following script enhances/overrides built-in checks, performs cross-field checks (e.g. confirm password), validates checkboxes group, file size/type, and sets custom messages.

```
// attach listener (better than inline onsubmit)
document.getElementById('fullForm').addEventListener('submit', function
    (e) {
  const form = e.target;
  let valid = true;
  let messages = [];

  // 1. Let browser run built-in validation first
  if (!form.checkValidity()) {
    // If built-in HTML5 constraints fail, allow browser to show its
       messages.
    // But we can also call reportValidity() to show messages
       programmatically:
```

```
    form.reportValidity();
    e.preventDefault();
    return;
  }

  // 2. Custom checks: password confirmation
  const pwd = form.querySelector('input[name="pwd"]');
  const confirmPwd = form.querySelector('input[name="confirmPwd"]'); //
      assume exists
  if (pwd && confirmPwd && pwd.value !== confirmPwd.value) {
    valid = false;
    confirmPwd.setCustomValidity('Passwords do not match');
    messages.push('Passwords do not match');
  } else if (confirmPwd) {
    confirmPwd.setCustomValidity(''); // clear previous message
  }

  // 3. Checkbox group: require at least one hobby
  const hobbies = form.querySelectorAll('input[name="hobby"]:checked');
  if (hobbies.length === 0) {
    valid = false;
    messages.push('Please select at least one hobby');
    // we can set custom validity on one hidden field or on a visible
        element
  }

  // 4. File size/type check
  const resume = form.querySelector('input[name="resume"]');
  if (resume && resume.files.length > 0) {
    const file = resume.files[0];
    const maxBytes = 2 * 1024 * 1024; // 2 MB
    if (file.size > maxBytes) {
      valid = false;
      resume.setCustomValidity('Resume must be smaller than 2 MB');
      messages.push('Resume too large (max 2 MB)');
    } else {
      resume.setCustomValidity('');
    }
    // accept attribute already restricts type, but additional checks
        are possible:
    const allowed = ['application/pdf'];
    if (!allowed.includes(file.type) && !file.type.startsWith('image/')
        ) {
      valid = false;
      resume.setCustomValidity('Only PDF or images are allowed');
      messages.push('Invalid resume type');
```

```
    }
  }

  // 5. If any custom validations failed, prevent submit and show
     messages
  if (!valid) {
    e.preventDefault();
    // Prefer browser's validity UI. If you want a custom UI, show
       messages:
    alert('Form validation failed:\\n' + messages.join('\\n'));
    // Optionally, call reportValidity to show the first custom message
       :
    form.reportValidity();
  }
});
```

5.1.4.4 *Notes on UX and validation strategy*

- Use **HTML5 built-in validation** for simple rules (required, type, min/max, pattern) — it is lightweight and consistent.
- Use **JavaScript custom validation** for cross-field logic, file size checks, async checks (e.g., username availability), and to display polished inline error UIs.
- Keep error messages clear and actionable. Use `title` with `pattern`, and `setCustomValidity()` to integrate custom messages with browser UI.
- Always replicate critical validations on the server (never trust client input).

5.2. Introduction to MERN

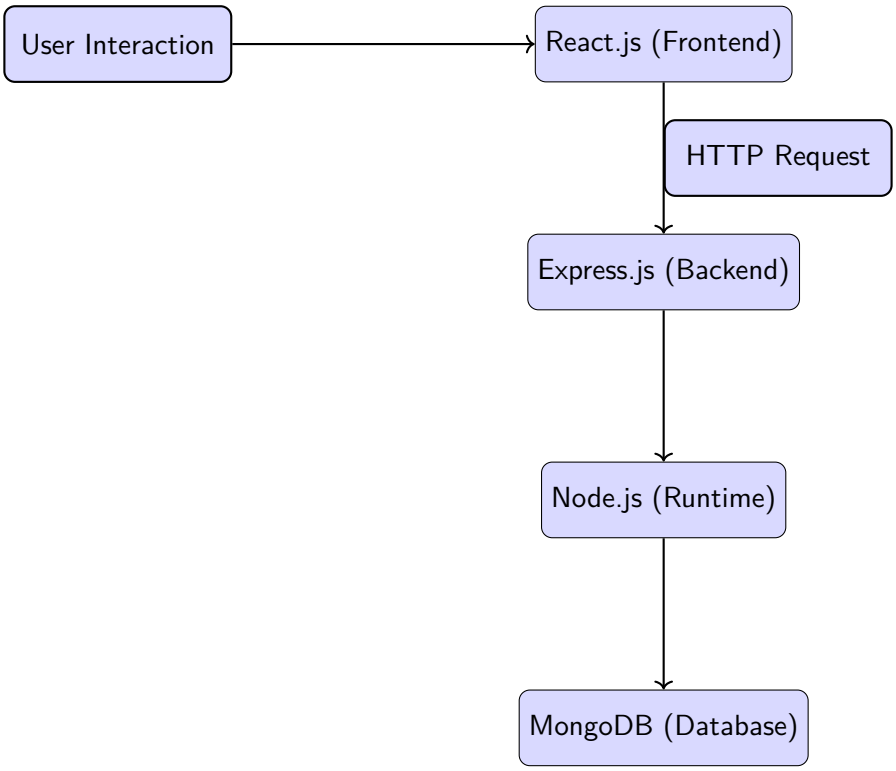
The MERN stack is a popular JavaScript-based technology stack for full-stack web development. It allows developers to use a single language (**JavaScript**) across the entire development process — database, backend, frontend, and runtime.

5.2.1 MERN Components

- **MongoDB** – A NoSQL database that stores data as flexible JSON-like documents. Suitable for applications with dynamic schemas and large-scale data.
- **Express.js** – A lightweight and minimal backend web framework for Node.js. Provides routing, middleware, and simplified API handling.
- **React.js** – A frontend library developed by Facebook for building user interfaces. It allows developers to build reusable UI components and handle state efficiently.
- **Node.js** – A JavaScript runtime environment built on Chrome's V8 engine. It executes JavaScript outside the browser and provides a scalable server-side platform.

5.2.2 MERN Workflow

- **Client** (React) sends requests for data.
- **Server** (Express + Node.js) processes requests and communicates with the database.
- **Database** (MongoDB) stores and retrieves data.
- The response is sent back to the React frontend for rendering.



5.3. Serverless Hello World Example

Serverless architecture allows us to run our code on the cloud without worrying about setting up or maintaining servers. The cloud provider (like AWS, Netlify, or Vercel) automatically handles scaling, availability, and billing — you only pay for the time your function runs.

5.3.1 Key Features of Serverless Computing

- **No server setup:** You focus on writing code, not on managing servers.
- **Event-driven:** Functions run only when triggered (e.g., HTTP request, database change, or file upload).
- **Auto-scaling:** Cloud platforms automatically adjust the number of running instances based on demand.
- **Pay-as-you-go:** You pay only when your function runs — no cost when idle.

5.3.1.1 *How It Works (Conceptually)*

Imagine you write a small JavaScript function. Whenever someone visits a specific URL (like `https://example.com/hello`), the cloud runs that function, returns a response, and then shuts down automatically.

5.3.2 Step 1: Writing a Serverless Function

Create a new file called `hello.js` and add the following code:

Listing 5.23: Simple Serverless Function

```
exports.handler = async (event) => {
  return {
    statusCode: 200,
    body: "Hello World from Serverless!"
  };
};
```

This code defines a small function that returns the text "Hello World from Serverless!" whenever someone calls it.

5.3.3 Step 2: Deploying on AWS Lambda

1. Sign in to the **AWS Management Console**.
2. Open **AWS Lambda** → Click **Create Function**.
3. Choose **Author from Scratch**.
4. Name your function (e.g., `HelloWorld`) and choose runtime **Node.js**.
5. Paste the code from `hello.js` into the editor.
6. Click **Deploy**.
7. Add an **API Gateway Trigger** to make your function available through a public HTTP URL.

5.3.4 Step 3: Testing the Function

After deployment, AWS gives you a test URL. Use your browser or a terminal command to test it:

```
curl https://<your-api-endpoint>.amazonaws.com/hello
# Output: Hello World from Serverless!
```

5.3.5 Why Serverless is Useful in MERN Projects

- **Simpler backend:** You don't need to run a full Express server.
- **Easy React integration:** The frontend can directly call Lambda URLs.
- **Scalable:** Automatically handles high traffic.
- **Cost-effective:** Ideal for small apps, startups, or student projects.

5.4. Netlify Serverless Hello World Project

Netlify is a beginner-friendly platform that lets you deploy static websites and serverless functions together — ideal for MERN or JAMstack projects.

5.4.1 Project Folder Structure

```
netlify-hello/
 index.html
 netlify/
     functions/
          hello.js
```

5.4.2 Step 1: Create the Frontend (index.html)

Listing 5.24: index.html

```
<!DOCTYPE html>
<html>
<head><title>Netlify Hello</title></head>
<body>
  <h1>Hello from Netlify!</h1>
  <button onclick="callApi()">Call Serverless Function</button>

  <script>
    async function callApi() {
      const res = await fetch("/.netlify/functions/hello");
      const data = await res.text();
      alert(data);
    }
  </script>
</body>
</html>
```

5.4.3 Step 2: Create the Serverless Function (hello.js)

Listing 5.25: netlify/functions/hello.js

```
exports.handler = async function(event, context) {
  return {
```

```
    statusCode: 200,
    body: "Hello World from Netlify Serverless!"
  };
};
```

5.4.4 Step 3: Install and Test Locally

```
npm install -g netlify-cli
cd netlify-hello
netlify dev
```

Then open http://localhost:8888. When you click the button, you'll see an alert box showing:

Hello World from Netlify Serverless!

5.4.5 Step 4: Deploying to Netlify

Option A — Manual Upload

1. Visit https://app.netlify.com.
2. Click **Add new site → Deploy manually**.
3. Drag your project folder (`netlify-hello/`) into the upload box.
4. Netlify gives you a live URL like: https://your-site-name.netlify.app.

Option B — GitHub Integration

1. Push your project to GitHub.
2. On Netlify, choose **Import from Git**.
3. Select your repository → click **Deploy**.
4. Every future Git push automatically redeploys your site.

5.5. Deploying a GitHub Project to Netlify

5.5.1 Step 1: Push Code to GitHub

```
git init
git add .
git commit -m "Initial commit"
```

```
git branch -M main
git remote add origin https://github.com/<username>/<repo>.git
git push -u origin main
```

5.5.2 Step 2: Connect Netlify with GitHub

1. Log in to https://app.netlify.com.
2. Click **Add new site → Import from Git**.
3. Choose **GitHub** and authorize access.
4. Select your project repository.

5.5.3 Step 3: Build Settings

- **Branch:** `main`
- **Build command:**
  - Blank for HTML
  - `npm run build` for React projects
- **Publish directory:**
  - `.` for HTML
  - `build` for React

Click **Deploy site**. In a few seconds, your site will be live.

5.6. Popular Serverless Platforms (Summary Table)

Platform	Languages Supported	Free Tier	Typical Uses
AWS Lambda	Node.js, Python, Java, Go, C#, Ruby	1M requests/month free	Enterprise apps, microservices, IoT
Google Cloud Functions	Node.js, Python, Go, Java, .NET	2M requests/month free	Firebase integration, analytics apps
Azure Functions	Node.js, Python, C#, Java	1M requests/month free	Microsoft integrations, corporate apps
IBM Cloud Functions	Node.js, Swift, Python, PHP	400k GB-seconds free/month	Academic and AI projects
Netlify Functions	Node.js, Go	125k requests/month	Static + dynamic web apps, JAMstack
Vercel Functions	Node.js, Go, Python, Ruby	125k requests/month	React/Next.js and frontend-first apps

Cloudflare Workers	JavaScript, Rust (WASM)	100k requests/day	Low-latency edge apps, CDN logic
OpenFaaS	Any (Docker-based)	Self-hosted (no cloud fee)	On-premise or learning FaaS concepts
Apache OpenWhisk	Node.js, Python, Java, Swift	Community edition free	Educational and open-source projects

5.7. Introduction to React and Component Basics

React is a popular JavaScript library used to build dynamic and interactive user interfaces. It allows developers to create web pages using **components**, which are small, reusable building blocks of a webpage.

5.7.1 What is a Component?

A **component** in React is like a small piece of the user interface (UI). For example, a button, navigation bar, or form can each be written as a component.

Each component:

- Has its own logic and appearance.
- Can be reused in multiple pages.
- Can be combined to form a complete web application.

Types of Components:

1. **Function Components** – simple JavaScript functions that return HTML (via JSX).
2. **Class Components** – ES6 classes with additional features like lifecycle methods and state.

Example of a simple **Function Component**:

Listing 5.26: Function Component Example

```
function Welcome() {
  return <h2>Hello from React Component!</h2>;
}
```

Example of a **Class Component**:

Listing 5.27: Class Component Example

```
class Welcome extends React.Component {
  render() {
    return <h2>Hello from Class Component!</h2>;
  }
}
```

5.8. Understanding JSX Syntax

JSX (JavaScript XML) is the special syntax used by React to write user interface code. It allows developers to write HTML-like tags directly inside JavaScript. JSX makes React code easy to read, understand, and maintain.

5.8.1 Why JSX is Used

Without JSX, we would have to write UI elements using plain JavaScript functions such as:

```
const element = React.createElement("h1", null, "Hello, world!");
```

This is valid React code, but it looks complex. JSX simplifies it to:

```
const element = <h1>Hello, world!</h1>;
```

Both examples produce the same result — a heading that says "Hello, world!". JSX is easier to read and write, which is why developers prefer it.

5.8.2 Basic JSX Rules

- JSX code must have **one parent element**. For example, you cannot return two tags side by side without wrapping them inside a single container such as a `<div>` or `<>...</>`.
- All HTML tags must be properly **closed**. Example: `<img />` instead of `<img>`.
- You can include JavaScript expressions inside curly braces `{}`.
- JSX attributes (like `class`, `for`) use JavaScript-style names: `className`, `htmlFor`, etc.

5.8.2.1 *Example 1: Simple JSX Element*

Listing 5.28: Simple JSX Example

```
const element = <h2>Welcome to React!</h2>;
ReactDOM.render(element, document.getElementById('root'));
```

Output:

Welcome to React!

5.8.2.2 *Example 2: Using JavaScript Expressions in JSX*

You can insert any valid JavaScript expression between `{}`.

Listing 5.29: JSX with JavaScript Expression

```
const name = "Ravi";
const city = "Hyderabad";
const element = <h3>Hello, {name} from {city}!</h3>;
```

```
ReactDOM.render(element, document.getElementById('root'));
```

Output:

Hello, Ravi from Hyderabad!

Here, the values of variables `name` and `city` are dynamically inserted into the HTML.

5.8.2.3 *Example 3: Embedding Functions Inside JSX*

You can also call functions inside JSX expressions.

Listing 5.30: JSX Calling a Function

```
function greet(user) {
  return "Hello, " + user + "!";
}

const element = <h1>{greet("Ananya")}</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Output:

Hello, Ananya!

This example shows that JSX can easily use JavaScript logic, making the UI dynamic and interactive.

5.8.2.4 *Example 4: Multiple Lines in JSX*

When JSX spans multiple lines, wrap it inside parentheses `( ... )` to improve readability and avoid syntax errors.

Listing 5.31: Multi-line JSX Example

```
const element = (
  <div>
    <h1>Welcome!</h1>
    <p>This is a multi-line JSX example.</p>
  </div>
);
ReactDOM.render(element, document.getElementById('root'));
```

Output:

Welcome! This is a multi-line JSX example.

5.8.2.5 *Example 5: Adding Attributes in JSX*

You can add attributes just like HTML, but in JSX they follow **camelCase** naming.

Listing 5.32: JSX Attributes Example

```
const element = (
  <img
    src="https://reactjs.org/logo-og.png"
    alt="React Logo"
    width="200"
  />
);
ReactDOM.render(element, document.getElementById('root'));
```

Output: An image of the React logo appears on the webpage.

Important Notes:

- Use `className` instead of `class`.
- Use `htmlFor` instead of `for`.
- Boolean attributes like `disabled`, `checked`, or `required` can be written as `disabled={true}`.

5.8.2.6 *Example 6: Conditional Rendering in JSX*

You can use conditions and operators (like ternary `?:`) inside JSX.

Listing 5.33: Conditional Rendering Example

```
const isLoggedIn = true;
const element = (
  <div>
    {isLoggedIn ? <h2>Welcome back!</h2> : <h2>Please log in.</h2>}
  </div>
);
ReactDOM.render(element, document.getElementById('root'));
```

Output:

Welcome back!

Changing `isLoggedIn` to `false` will display:

Please log in.

5.8.3 JSX vs HTML: Key Differences

Feature	HTML	JSX
Syntax Type	Markup Language	JavaScript Extension (XML style)
Attributes	`class`, `for`	`className`, `htmlFor`
Expressions	Static content only	Dynamic with {}
Closing Tags	Optional for some tags	Mandatory for all tags
Variables	Not allowed inside tags	Allowed using {}

5.8.4 Common JSX Mistakes by Beginners

- Forgetting to use `className` instead of `class`.
- Returning multiple elements without a parent wrapper.
- Not closing tags like `<img>` or `<input>`.
- Missing parentheses around multi-line JSX.
- Writing statements (like `if`) directly inside JSX instead of using expressions.

5.8.5 Exercise: Practice JSX

Task 1: Personalized Greeting

```
const name = "Kiran";
const age = 20;
const message = <p>{name} is {age} years old.</p>;
ReactDOM.render(message, document.getElementById('root'));
```

Output: Kiran is 20 years old.

Task 2: Multi-line JSX Card

```
const student = (
  <div style={{border: "1px solid gray", padding: "10px"}}>
    <h2>Student Details</h2>
    <p>Name: Arjun</p>
    <p>Course: BCA</p>
  </div>
);
ReactDOM.render(student, document.getElementById('root'));
```

Try This:

- Change the data (name, course).
- Add another field (college, year).
- Change border color and text color using inline styles.

5.8.6 Summary of JSX Concepts

- JSX allows HTML-like syntax inside JavaScript.
- Expressions inside {} make UI dynamic.
- Every JSX block must have a single parent tag.
- All elements must be closed properly.
- JSX improves readability and reduces code complexity.

Key Point: JSX is not HTML — it is JavaScript with a more readable syntax that helps build rich,

interactive UIs efficiently.

5.8.7 Rendering Components using ReactDOM

React components are displayed (or **rendered**) inside an HTML page using the `ReactDOM.render()` function.

Listing 5.34: Rendering React Component

```
const rootElement = document.getElementById('root');
ReactDOM.render(<Welcome />, rootElement);
```

Here:

- `Welcome` is the React component.
- `root` is a div in `index.html` where React attaches the app.

5.8.8 Folder Structure of a React Project

After creating a new React project, you will see this folder layout:

Listing 5.35: Typical React Folder Structure

```
issue-tracker/
 node_modules/
 public/
     index.html
 src/
     App.js
     index.js
     components/
 package.json
```

Explanation:

- **src/** – contains all your JavaScript and React components.
- **public/** – holds static files (like `index.html`).
- **index.js** – the entry file that loads the first React component.
- **App.js** – the main component where the app begins.

5.8.9 Step-by-Step Demo: Create a React App

Step 1: Install `Node.js` (if not already installed). You can verify installation:

```
node -v
npm -v
```

Step 2: Create a new React project using `create-react-app`:

Listing 5.36: Creating React App

```
npx create-react-app issue-tracker
cd issue-tracker
npm start
```

This command automatically sets up everything required for React development and opens the app in a browser at http://localhost:3000.

5.8.10 Step 3: Understanding the App Component

Open `src/App.js`. By default, it looks like this:

Listing 5.37: Default App Component

```
function App() {
  return (
    <div className="App">
      <h1>Hello World</h1>
    </div>
  );
}
export default App;
```

Explanation:

- `function App()` defines a function component.
- `return (...)` tells what to display on the webpage.
- `export default App;` allows other files to import this component.

When you run `npm start`, React automatically displays this on your web browser.

5.8.11 Step 4: Modifying Component Text and Styles

Students can customize the `App.js` component.

Listing 5.38: Customized App Component

```
function App() {
  const title = "Welcome to React!";
  return (
    <div style={{ textAlign: "center", color: "blue" }}>
      <h1>{title}</h1>
      <p>This is your first React component.</p>
    </div>
  );
}
export default App;
```

Try this:

- Change the color or font style.
- Add another heading or paragraph.
- Add an image using the `<img />` tag.

5.8.12 Exercise for Students

1. Create a new React project named `my-first-react-app`.
2. Open `App.js` and change the message to display your name.
3. Add a new paragraph or image using JSX.
4. Use inline CSS styles (like color, fontSize) to decorate your message.
5. Save and check your output at http://localhost:3000.

5.8.13 Summary

- React builds web pages using small reusable components.
- JSX allows you to write HTML directly in JavaScript.
- Components are rendered using `ReactDOM.render()`.
- Each React project follows a simple folder structure.
- The first step is always creating and modifying the `App` component.

Key Takeaway: In React, everything is a component. By combining multiple components, we can create complex and interactive web applications.

5.9. React Classes vs Functional Components

React applications can be created using two main types of components — **Class Components** and **Functional Components**. Both serve the same purpose: building reusable pieces of a user interface. However, they differ in their **syntax**, **structure**, and the way they **manage data (state)** and **respond to lifecycle events**. In this chapter, we will develop a simple React project that demonstrates both component types side by side, helping you understand their similarities and differences in a practical way.

5.9.1 Goal of This topic

- Understand Class and Functional Components.
- Learn how to use `state` and `hooks`.
- See how both display the same output.
- Learn when to use each type in real projects.

5.9.2 Project Overview

We will build a small React app called **component-demo** that contains:

React Component Lifecycle (Class-based)

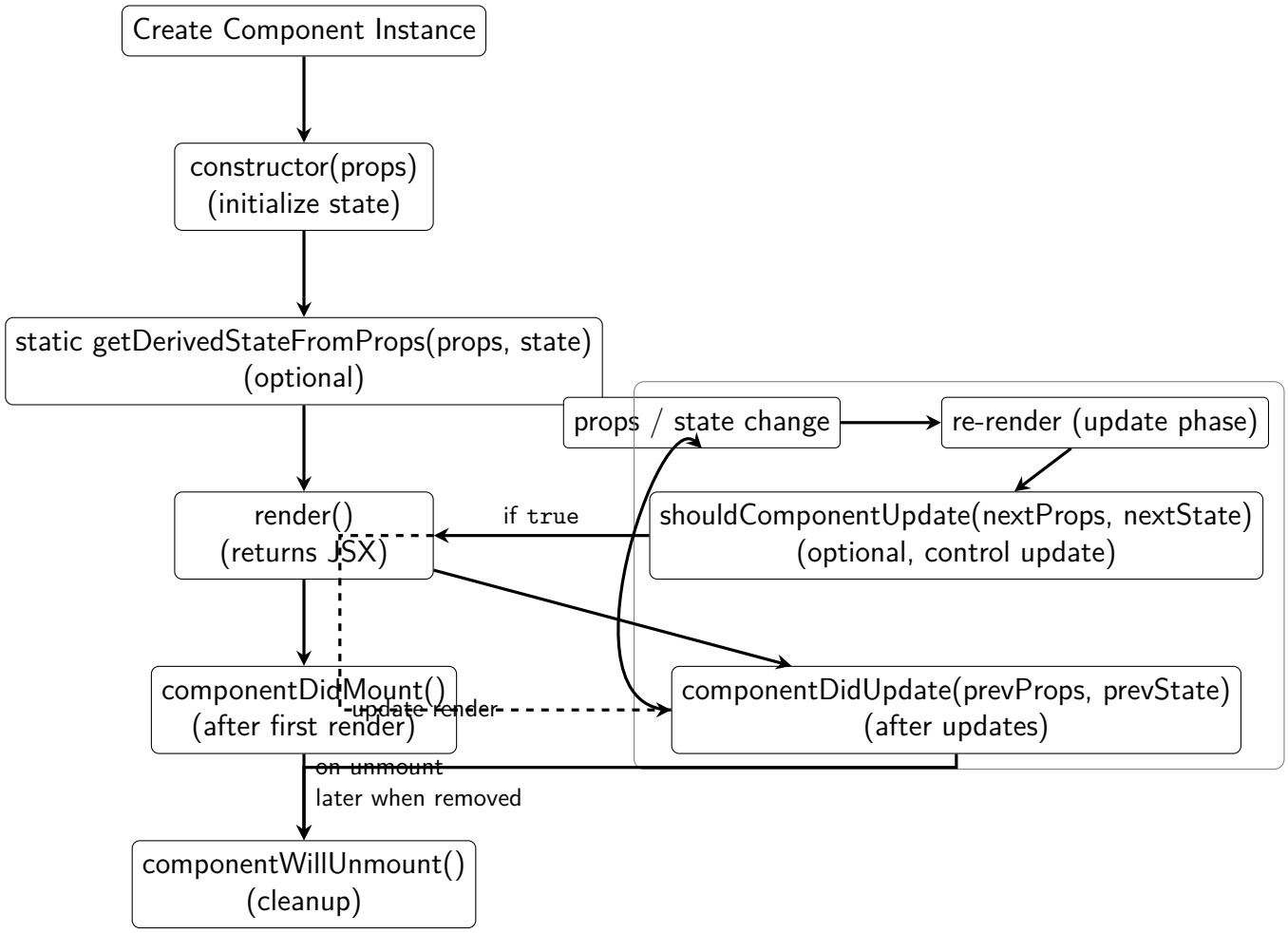


Figure 5.1: Typical lifecycle of a React class component with update flow and the hooks mapping.

- A Class Component (`CounterClass.js`)
- A Functional Component (`CounterFunction.js`)

Both components will show a counter that increases when a button is clicked.

5.9.3 Step 1: Create a New React Project

Open your terminal or command prompt and run the following commands:

Listing 5.39: Creating a new React project

```
npx create-react-app component-demo
cd component-demo
npm start
```

Explanation:

- `npx create-react-app component-demo` → Creates a new React project with all dependencies.
- `cd component-demo` → Moves into the project folder.

- `npm start` → Starts the local development server.

  Once started, your default React app will open automatically in the browser at:

  `http://localhost:3000`

  You should see the "React Logo" page.

5.9.4 Step 2: Project Folder Structure

After creating the project, your folder will look like this:

Listing 5.40: React Project Folder Structure

```
component-demo/
 node_modules/
 public/
     index.html
 src/
     App.js
     index.js
     components/
          CounterClass.js
          CounterFunction.js
 package.json
 README.md
```

We will write our code inside the `src/components/` folder. If the `components` folder doesn't exist, create it manually.

5.9.5 Step 3: Create a Class Component

Create a new file inside the `src/components/` folder named `CounterClass.js`.

Listing 5.41: src/components/CounterClass.js

```
import React, { Component } from "react";

class CounterClass extends Component {
  constructor() {
    super();
    this.state = { count: 0 }; // initial state
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
```

```
    return (
      <div style={{ textAlign: "center", marginTop: "30px" }}>
        <h2>Class Component Counter</h2>
        <h3>Count: {this.state.count}</h3>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default CounterClass;
```

Explanation:

- `this.state` stores the count value.
- `setState()` updates the value.
- Each time the button is clicked, the count increases by one.

5.9.6 Step 4: Create a Functional Component

Create another file named `CounterFunction.js` inside the same `src/components/` folder.

Listing 5.42: src/components/CounterFunction.js

```
import React, { useState } from "react";

function CounterFunction() {
  const [count, setCount] = useState(0); // useState Hook

  return (
    <div style={{ textAlign: "center", marginTop: "30px" }}>
      <h2>Functional Component Counter</h2>
      <h3>Count: {count}</h3>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default CounterFunction;
```

Explanation:

- `useState(0)` creates a variable count starting at 0.
- `setCount()` updates it.
- This performs the same function as the class-based version, but with fewer lines of code.

5.9.7 Step 5: Display Both Components in App.js

Open the main file `src/App.js` and replace its contents with:

Listing 5.43: src/App.js

```
import React from "react";
import CounterClass from "./components/CounterClass";
import CounterFunction from "./components/CounterFunction";

function App() {
  return (
    <div>
      <h1 style={{ textAlign: "center", color: "blue" }}>
        React Components: Class vs Function
      </h1>
      <CounterClass />
      <CounterFunction />
    </div>
  );
}

export default App;
```

5.9.8 Step 6: Run the Application

In the terminal, ensure you are inside the project folder and run:

```
npm start
```

This starts the development server and opens your app in a browser.
You should see:

- A blue heading: "React Components: Class vs Function"
- A **Class Component Counter** with a button
- A **Functional Component Counter** with a button

Both buttons work independently and update their own counters.

5.9.9 Step 7: Understanding the Output

When you click the button under each counter:

- The **Class Component** uses `this.setState()` to update the count.
- The **Functional Component** uses the `useState()` Hook to do the same.

This shows that both methods achieve identical functionality with different syntax styles.

5.9.10 Step 8: Adding useEffect (Lifecycle Simulation)

To make the Functional Component behave like a Class Component's lifecycle, add the `useEffect()` Hook.

Update `CounterFunction.js` as follows:

Listing 5.44: useEffect Example in Functional Component

```
import React, { useState, useEffect } from "react";

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

  useEffect(() => {
    console.log("Component rendered or count changed!");
  }, [count]); // runs every time 'count' changes

  return (
    <div style={{ textAlign: "center", marginTop: "30px" }}>
      <h2>Functional Component Counter</h2>
      <h3>Count: {count}</h3>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default CounterFunction;
```

Now every time the count value updates, a message appears in the browser console.

5.9.11 Step 9: Class vs Functional Comparison Table

Feature	Class Component	Functional Component
Syntax	Uses ES6 Class and `render()` method	Uses simple function syntax
State Handling	Uses `this.state` and `this.setState()`	Uses `useState()` Hook
Lifecycle	Uses lifecycle methods	Uses `useEffect()` Hook
Code Length	Longer and verbose	Short and easy to read
Ease of Use	Requires understanding of `this` keyword	Easier for beginners
Modern Practice	Traditional (pre-React 16.8)	Recommended (post React 16.8)

5.9.12 Step 10: Exercise for Students

Task 1: Add a "Decrement" button to both counters.
Task 2: Add a "Reset" button that sets count back to 0.
Task 3: Display a message below the count:

"You clicked X times"

Task 4: Convert the Class Component (`CounterClass.js`) into a Functional Component using Hooks.

5.9.13 Summary

- Class Components use `state`, `setState()`, and lifecycle methods.
- Functional Components use Hooks like `useState()` and `useEffect()`.
- Functional Components are shorter, cleaner, and preferred in modern React.
- Both can achieve the same output — it's just a difference in writing style.

Key Takeaway: Today, most React developers use **Functional Components with Hooks** because they are easier to maintain and more efficient.

5.10. Composing Components in React

In React, large applications are built by combining many small, reusable components. This process is called **component composition**. Each component focuses on a single task, and when combined, they form the complete user interface (UI).

5.10.1 Goal of This Class

- Understand the concept of composing multiple components.
- Learn how components communicate as **parent** and **child**.
- Build a small **Issue Tracker** interface using separate components.
- Practice rendering multiple components in `App.js`.

5.10.2 Why Compose Components?

Composition means building complex UIs by joining simple parts together.

- Each part (component) can be reused in other pages or projects.
- Small components are easier to debug, test, and maintain.
- They make your project well organized and scalable.

Example Idea: Instead of writing one big component for an entire page, we can divide it into:

- `IssueFilter` – filters issues by status.
- `IssueList` – displays issues in a table.
- `IssueAdd` – adds a new issue.

5.10.3 Parent–Child Relationship and Component Tree

In React, components can be nested. A parent component can include one or more child components.

`App (Parent)  IssueFilter (Child)  IssueList (Child)  IssueAdd (Child)`

This structure forms a **component tree**. The parent passes data to children using `props`.

5.10.4 Step 1: Create the React Project

Open the terminal and create a new project named `issue-tracker`.

Listing 5.45: Creating a new project

```
npx create-react-app issue-tracker
cd issue-tracker
npm start
```

The default React app will start automatically at http://localhost:3000.

5.10.5 Step 2: Organize Folder Structure

Inside the `src/` folder, create a subfolder called `components/`. Your project should look like this:

Listing 5.46: Folder structure

```
issue-tracker/
 src/
    App.js
    index.js
    components/
        IssueFilter.js
        IssueList.js
        IssueAdd.js
 public/
 package.json
```

5.10.6 Step 3: Create the IssueFilter Component

Listing 5.47: src/components/IssueFilter.js

```
import React from "react";

function IssueFilter() {
  return (
    <div style={{ marginBottom: "20px" }}>
      <h3>Issue Filter</h3>
      <select>
        <option>All</option>
        <option>Open</option>
```

```
        <option>Closed</option>
      </select>
    </div>
  );
}

export default IssueFilter;
```

Explanation:

- A simple dropdown to filter issues by status.
- Later, you can connect it with real filtering logic.

5.10.7 Step 4: Create the IssueList Component

Listing 5.48: src/components/IssueList.js

```
import React from "react";

function IssueList() {
  const issues = [
    { id: 1, title: "Login bug", status: "Open" },
    { id: 2, title: "UI glitch", status: "Closed" },
  ];

  return (
    <div>
      <h3>Issue List</h3>
      <table border="1" cellPadding="8" style={{ width: "100%",
         textAlign: "left" }}>
        <thead>
          <tr>
            <th>ID</th>
            <th>Title</th>
            <th>Status</th>
          </tr>
        </thead>
        <tbody>
          {issues.map((issue) => (
            <tr key={issue.id}>
              <td>{issue.id}</td>
              <td>{issue.title}</td>
              <td>{issue.status}</td>
            </tr>
          ))}
        </tbody>
```

```
      </table>
    </div>
  );
}

export default IssueList;
```

Explanation:

- The issues are stored in an array.
- The `map()` function creates a table row for each issue.
- `key` helps React identify each element.

5.10.8 Step 5: Create the IssueAdd Component

Listing 5.49: src/components/IssueAdd.js

```
import React, { useState } from "react";

function IssueAdd() {
  const [title, setTitle] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    alert("New issue added: " + title);
    setTitle("");
  };

  return (
    <div style={{ marginTop: "20px" }}>
      <h3>Add New Issue</h3>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          placeholder="Enter issue title"
        />
        <button type="submit">Add</button>
      </form>
    </div>
  );
}

export default IssueAdd;
```

Explanation:

- `useState()` manages the input field's value.
- When the form is submitted, it shows an alert message.

5.10.9 Step 6: Combine All Components in App.js

Open `src/App.js` and replace its content with the following:

Listing 5.50: src/App.js

```
import React from "react";
import IssueFilter from "./components/IssueFilter";
import IssueList from "./components/IssueList";
import IssueAdd from "./components/IssueAdd";

function App() {
  return (
    <div style={{ width: "80%", margin: "auto" }}>
      <h1 style={{ textAlign: "center", color: "teal" }}>Issue Tracker
         </h1>
      <IssueFilter />
      <IssueList />
      <IssueAdd />
    </div>
  );
}

export default App;
```

Explanation:

- `App.js` acts as the parent component.
- It imports and renders all three child components.
- Each child handles a different part of the interface.

5.10.10 Step 7: Run the Application

Start the project using:

```
npm start
```

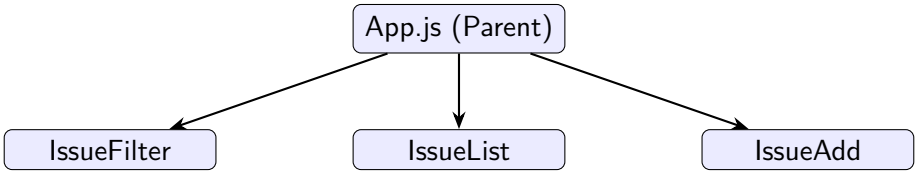
Your application will open in the browser. You should see:

1. A title "Issue Tracker".
2. A dropdown filter for issues.
3. A table showing two issues.

4. A form to add new issues.

When you type a title and click "Add", an alert message will display.

5.10.11 Understanding the Component Tree



Explanation: The parent component (`App.js`) controls the layout and renders three children. Each child performs one specific function — filtering, displaying, or adding issues.

5.10.12 Step 8: Exercises for Students

1. Add another field in `IssueAdd.js` for "Status" (Open / Closed).
2. Display the new issue dynamically in `IssueList`.
3. Style the table with colors and borders using inline CSS.
4. Create a `Header.js` and `Footer.js` component and include them in `App.js`.

5.10.13 Summary

- Large UIs can be built easily by composing smaller, reusable components.
- Parent components can render multiple child components.
- Data and events can be passed between components using `props`.
- Composition makes your React application modular and maintainable.

5.10.14 Exercise: Compose Your Own Layout (Header / Main /›Footer)

Goal: Students will practice composing a simple page layout by building three reusable components: `Header`, `Main` and `Footer`. This exercise reinforces component composition, props, basic styling and project structure.

Instructions (step-by-step)

1. Inside an existing React project (or create a new one with `npx create-react-app my-layout`), open the `src/` folder and create a subfolder named `components/`.
2. Create three files:
   - `src/components/Header.js`
   - `src/components/Main.js`
   - `src/components/Footer.js`

3. Optionally create a CSS file `src/components/Layout.css` for shared styles.
4. Update `src/App.js` to import and render these components.
5. Run the app with `npm start` and verify the layout in the browser.

Suggested Folder Structure

```
my-layout/
 src/
     App.js
     index.js
     components/
         Header.js
         Main.js
         Footer.js
         Layout.css    # optional
```

Example Component Code

5.10.14.1 *Header.js*

Listing 5.51: src/components/Header.js

```
import React from "react";
import "./Layout.css"; // optional shared styles

function Header({ title, subtitle }) {
  return (
    <header className="site-header">
      <h1>{title}</h1>
      {subtitle && <p className="subtitle">{subtitle}</p>}
      <nav>
        <a href="#home">Home</a> | <a href="#about">About</a> | <a href
           ="#contact">Contact</a>
      </nav>
    </header>
  );
}

export default Header;
```

5.10.14.2 *Main.js*

Listing 5.52: src/components/Main.js

```
import React from "react";

function Main({ children }) {
  return (
    <main style={{ padding: "16px", minHeight: "300px" }}>
      {children ? children : (
        <>
          <h2>Welcome</h2>
          <p>This is the main content area. Replace this with your
             content.</p>
        </>
      )}
    </main>
  );
}

export default Main;
```

5.10.14.3 *Footer.js*

Listing 5.53: src/components/Footer.js

```
import React from "react";

function Footer({ copyright }) {
  return (
    <footer style={{ textAlign: "center", padding: "12px 0", borderTop:
        "1px solid #ddd" }}>
      <small>{copyright}</small>
    </footer>
  );
}

export default Footer;
```

5.10.14.4 *Optional Layout.css*

Listing 5.54: src/components/Layout.css

```
.site-header {
  background: #0d6efd;
  color: white;
  padding: 16px;
  text-align: center;
```

```
}
.site-header .subtitle {
  margin: 4px 0 8px 0;
  font-weight: 400;
  opacity: 0.9;
}
.site-header nav a {
  color: rgba(255,255,255,0.9);
  text-decoration: none;
  margin: 0 6px;
}
```

App.js (compose the layout)
Replace the contents of `src/App.js` with:

Listing 5.55: src/App.js

```
import React from "react";
import Header from "./components/Header";
import Main from "./components/Main";
import Footer from "./components/Footer";
import "./components/Layout.css"; // optional

function App() {
  return (
    <div style={{ maxWidth: "900px", margin: "20px auto", boxShadow:"0
       0 6px rgba(0,0,0,0.08)", borderRadius: "4px", overflow: "hidden"
        }}>
      <Header title="My Website" subtitle="A small demo layout" />
      <Main>
        <h3>Today's Notes</h3>
        <p>Build components and pass data using props and children.</p>
      </Main>
      <Footer copyright={`© ${new Date().getFullYear()} Your Name`} />
    </div>
  );
}

export default App;
```

Commands to run

```
# if starting fresh
npx create-react-app my-layout
cd my-layout
# add the files above, then
```

```
npm start
```

Open http://localhost:3000 to view the composed layout.

Extensions and Variation Tasks (for students)

1. **Props practice:** Modify `Header` to accept a `links` prop (array of {label, href}) and render navigation dynamically.
2. **Children usage:** Use `Main` with different children in multiple pages (for example, a Home and About section).
3. **State lift-up:** Add a simple search box inside `Header` and lift state up to `App` to filter content displayed in `Main`.
4. **Styling:** Convert inline styles to a CSS module or styled-components (optional advanced).
5. **Responsive layout:** Add CSS to make the header/nav responsive (stack links vertically on small screens).
6. **Accessibility:** Ensure nav links have sensible text and keyboard focus; add `aria`-labels where appropriate.

Assessment Checklist (what students should submit)

- A working React app that renders `Header`, `Main`, and `Footer`.
- `Header` accepts and displays a `title` prop.
- `Main` demonstrates use of `children`.
- `Footer` shows the current year (or a prop).
- Bonus: one or more extension tasks completed.

Tip for instructors: Walk students through creating one component live, then let them pair-program the remaining two. Encourage saving changes and using the browser reload to see instant feedback.

Key Takeaway: Every React project should be broken down into small, reusable parts. This approach improves readability, reduces code repetition, and makes collaboration easier.

5.11. Passing Data Using Properties (Props)

React uses a concept called **props** (short for "properties") to send data from one component to another. Props allow components to be **dynamic**, **reusable**, and **customizable**. They follow a **one-way data flow** — data moves only from parent to child.

5.11.1 Goal of This Class

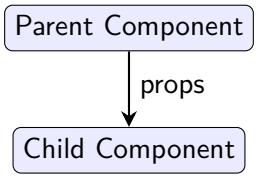
- Understand how data is passed from a parent to a child component using `props`.
- Learn the one-way data flow principle in React.
- Explore how props make components reusable.
- Practice reading props inside child components.

5.11.2 What are Props?

Props are inputs to React components. They behave just like function parameters. When a parent component wants to pass data to a child, it sends them through attributes.

Props are **read-only** — a child component cannot change them.

5.11.3 One-way Data Flow Concept



Explanation: Data always flows downward — from parent to child. If the child needs to communicate back, it can call a function passed from the parent as a prop.

5.11.4 Step 1: Create a New React Project

Listing 5.56: Creating a new React project

```
npx create-react-app props-demo
cd props-demo
npm start
```

Your browser will open automatically at `http://localhost:3000`.

5.11.5 Step 2: Folder Structure

Listing 5.57: Project folder structure

```
props-demo/
 src/
    App.js
    index.js
    components/
        Issue.js
```

We will pass data from `App.js` (parent) to `Issue.js` (child).

5.11.6 Step 3: Create the Child Component (Issue.js)

Listing 5.58: src/components/Issue.js

```
import React from "react";

function Issue(props) {
  return (
    <div style={{
      border: "1px solid gray",
      margin: "10px",
      padding: "10px",
      borderRadius: "6px",
      backgroundColor: "#f9f9f9"
    }}>
      <h3>Issue Title: {props.title}</h3>
      <p>Status: {props.status}</p>
    </div>
  );
}

export default Issue;
```

Explanation:

- The component Issue receives props as a parameter.
- The values of `props.title` and `props.status` are displayed.
- `props` acts like an object that holds all passed data.

5.11.7 Step 4: Send Props from the Parent (App.js) ›

Listing 5.59: src/App.js

```
import React from "react";
import Issue from "./components/Issue";

function App() {
  return (
    <div style={{ textAlign: "center" }}>
      <h1>Issue Tracker</h1>

      {/* Passing props to child components */}
      <Issue title="Login Bug" status="Open" />
      <Issue title="API Failure" status="Closed" />
      <Issue title="UI Misalignment" status="In Progress" />
    </div>
  );
}
```

```
export default App;
```

Explanation:

- The parent component App passes data as attributes inside the child tag.
- Each <Issue /> gets its own title and status.
- These values are automatically sent into the child component as props.

5.11.8 Step 5: Run the Project

Run the following command to view output:

```
npm start
```

Output in Browser:

Issue Tracker Issue Title: Login Bug — Status: Open Issue Title: API Failure — Status: Closed Issue Title: UI Misalignment — Status: In Progress

5.11.9 Understanding the Props Object

props is a simple JavaScript object. You can inspect it using console.log(props).

Listing 5.60: Inspecting props

```
function Issue(props) {
  console.log(props);
  return <h3>{props.title}</h3>;
}
```

In the browser console, you will see:

```
{title: "Login Bug", status: "Open"}
```

5.11.10 Step 6: Simplifying Using Object Destructuring

Instead of writing props.title and props.status, you can use destructuring directly in the parameter list.

Listing 5.61: Using destructuring for cleaner code

```
function Issue({ title, status }) {
  return (
    <div>
      <h3>{title}</h3>
      <p>Status: {status}</p>
```

```
    </div>
  );
}
```

This syntax makes the code shorter and easier to read.

5.11.11 Step 7: Props are Read-Only

Props cannot be modified inside the child component.

Listing 5.62: Incorrect usage example

```
function Issue(props) {
  props.title = "Changed Title"; //  Not allowed
  return <h3>{props.title}</h3>;
}
```

React will not allow direct changes to props because data must always flow downward. If data needs to change, it should be managed in the parent component's state.

5.11.12 Step 8: Exercise — Pass Student Info Using Props

Goal: Pass student details (name, course, and grade) from the parent component to a child component.
Instructions:

1. Create a new component called `Student.js` inside `src/components/`.
2. Use props to display:
   - Student Name
   - Course
   - Grade
3. In `App.js`, create three different `<Student />` components with different data.

Example Solution:

5.11.12.1 *Student.js*

Listing 5.63: src/components/Student.js

```
import React from "react";

function Student({ name, course, grade }) {
  return (
    <div style={{
      border: "1px solid #ccc",
      borderRadius: "6px",
      padding: "10px",
```

```
      margin: "10px",
      backgroundColor: "#f5faff"
    }}>
      <h3>{name}</h3>
      <p>Course: {course}</p>
      <p>Grade: {grade}</p>
    </div>
  );
}

export default Student;
```

5.11.12.2 *App.js*

Listing 5.64: src/App.js (using Student component)

```
import React from "react";
import Student from "./components/Student";

function App() {
  return (
    <div style={{ textAlign: "center" }}>
      <h1>Student Information</h1>

      <Student name="Ravi Kumar" course="BCA" grade="A+" />
      <Student name="Ananya Singh" course="BSc CS" grade="A" />
      <Student name="Rahul Verma" course="B.Tech IT" grade="B+" />
    </div>
  );
}

export default App;
```

Output:

Student Information Ravi Kumar — BCA — Grade: A+ Ananya Singh — BSc CS — Grade: A Rahul Verma — B.Tech IT — Grade: B+

5.11.13 Summary

- Props allow data to move from parent to child components.
- They make components flexible and reusable.
- Props are always read-only — a child should never modify them.
- The parent can send any kind of data (text, numbers, arrays, objects, or functions).

Key Takeaway: Props enable **component communication** in React and maintain a clean, predictable **one-way data flow**.

5.12. Passing Data Using children

React components receive data via `props`. One special prop is `children` — it contains whatever is placed between a component's opening and closing tags. Using `children` lets you build flexible, slot-like components (similar to `<slot>` in web components) that can wrap arbitrary content.

5.12.1 Goal of This Class

- Understand the difference between normal props and the special `children` prop.
- Learn how to create wrapper components (Card/Panel) that accept children.
- Practice using `children` to compose UIs in a flexible way.

5.12.2 Props vs children

- **Props:** named attributes passed on a component tag (e.g., `title="Hello"`). They are read-only values received by the child.
- **`children`:** everything placed between the opening and closing tags of a component. It can be text, elements, an expression, or even other components.

Analogy: Think of a `Card` component as a box with an empty space (slot). The parent puts content into that slot — the box (Card) decorates or wraps it.

5.12.3 Basic children Example

Listing 5.65: Using children in JSX

```
<Card>
  <h2>Issue Details</h2>
  <p>Some info about the issue.</p>
</Card>
```

Inside `Card`, the content above is available as `props.children`.

5.12.4 Step-by-step Demo (Card / Panel)

This demo builds a small React app demonstrating `children`. Follow these steps to create and run it.

5.12.4.1 *Create Project*

```
npx create-react-app children-demo
cd children-demo
npm start
```

5.12.4.2 *Folder Structure (inside src/)*

```
children-demo/
 src/
     App.js
     index.js
     components/
         Card.js
         Panel.js
```

5.12.4.3 *src/components/Card.js*

Listing 5.66: src/components/Card.js

```
import React from "react";

function Card({ children, style }) {
  const defaultStyle = {
    border: "1px solid #ddd",
    padding: "12px",
    borderRadius: "6px",
    background: "#fff",
    boxShadow: "0 1px 3px rgba(0,0,0,0.06)"
  };

  return (
    <div style={{ ...defaultStyle, ...style }}>
      {children}
    </div>
  );
}

export default Card;
```

Notes:

- children may be a single node or many nodes.
- We accept an optional style prop to allow parent styling.

5.12.4.4 *src/components/Panel.js*

Listing 5.67: src/components/Panel.js

```
import React from "react";
import Card from "./Card";
```

```
function Panel({ title, children }) {
  return (
    <Card style={{ marginBottom: "12px" }}>
      <header style={{ borderBottom: "1px solid #eee", paddingBottom: "
         8px", marginBottom: "8px" }}>
        <h3 style={{ margin: 0 }}>{title}</h3>
      </header>
      <div>
        {children}
      </div>
    </Card>
  );
}

export default Panel;
```

Notes:

- Panel composes Card and renders a title prop plus whatever is passed as children.
- This pattern separates decoration (Card) from semantic grouping (Panel).

5.12.4.5 *src/App.js (demo usage)*

Listing 5.68: src/App.js

```
import React from "react";
import Panel from "./components/Panel";
import Card from "./components/Card";

function App() {
  return (
    <div style={{ maxWidth: "760px", margin: "20px auto", fontFamily: "
       Arial, sans-serif" }}>
      <h1 style={{ textAlign: "center" }}>Children Prop Demo</h1>

      <Panel title="Issue Details">
        <h4>Issue #102: Login Bug</h4>
        <p>The login form throws an error when the username contains
           spaces.</p>
      </Panel>

      <Panel title="Quick Notes">
        <ul>
          <li>Use props for named configuration</li>
          <li>Use <code>children</code> for content slots</li>
        </ul>
```

```
      </Panel>

      <Card>
        <p>This is a plain card without a header. You can pass any JSX
           here.</p>
        <button onClick={() => alert("Card button clicked!")}>Click me
           </button>
      </Card>

    </div>
  );
}

export default App;
```

5.12.5 Run the App

After adding the files above, run:

```
npm start
```

Open http://localhost:3000 — you will see two Panels and a Card demonstrating children usage.

5.12.6 Advanced children Patterns

- **Single child vs multiple children:** props.children can be a single element, an array of elements, or text. Use React.Children utilities to work consistently.
- **Conditional children:** render children only when provided:

```
{children ? <div>{children}</div> : <div>No content</div>}
```

- **Cloning children (rare):** sometimes you may want to inject props into child elements:

```
React.Children.map(children, child =>
  React.isValidElement(child) ? React.cloneElement(child, {
     extraProp: value }) : child
)
```

Use this only when necessary — it can make code harder to read.

5.12.7 Exercise: Build a Reusable Panel/Card Component

Goal: Create a Panel component that accepts a title prop and arbitrary children. Use it to assemble a small dashboard.

Steps:

1. Create `Card.js` and `Panel.js` as shown above.
2. Build three different Panels in `App.js`:
   - `Panel title="Profile"` — include an image and profile text.
   - `Panel title="Tasks"` — list three tasks (use `<ul>`).
   - `Panel title="Actions"` — include two buttons.
3. Style the panels with simple inline styles or a CSS file.
4. Bonus: allow `Panel` to accept a `footer` prop (a node) and render it below the children.

Assessment checklist:

- Panel displays the `title`.
- Panel renders the children you pass.
- Panel can accept optional styling via props.
- Bonus: footer prop renders correctly.

5.12.8 Common Beginner Mistakes

- Trying to pass children via a named prop (e.g., `content={...}`) when the intended pattern is `children`. Both are valid, but `children` is idiomatic for slot-like content.
- Mutating `props.children` directly — treat children as read-only data.
- Not handling the case where `children` is `undefined`.

5.12.9 Summary

- `children` is a special prop that contains whatever is placed between a component's tags.
- Use `children` to build flexible wrapper components (Card, Panel, Modal).
- Combine named props (for configuration) with `children` (for content) to create powerful, reusable UI primitives.

5.13. Dynamic Composition in React

React allows you to build user interfaces dynamically by rendering components based on arrays or data structures. Instead of manually writing many component instances, you can generate them using JavaScript's `.map()` function. This process is known as **dynamic composition** — combining data and components at runtime.

5.13.1 Goal of This Class

- Learn how to render lists of components dynamically using `.map()`.
- Understand why `key` attributes are needed.
- Practice conditional rendering (show/hide based on data).

- Build a small dynamic issue list and a student list as exercises.

5.13.2 What is Dynamic Composition?

Dynamic composition means creating multiple components from data automatically. For example, instead of manually writing:

```
<Issue title="Bug 1" />
<Issue title="Bug 2" />
<Issue title="Bug 3" />
```

We can use an array:

```
const issues = ["Bug 1", "Bug 2", "Bug 3"];
{issues.map(title => <Issue title={title} />)}
```

5.13.3 The .map() Method in React

The .map() function in JavaScript is commonly used in React to loop through arrays and return JSX elements.

Syntax:

```
array.map((item, index) => {
  return <Component key={index} />;
});
```

Each returned component should have a unique key prop so that React can track which items changed, were added, or removed.

5.13.4 Step-by-Step Demo: Rendering an Issue List Dynamically

5.13.4.1 *1. Create a Project*

```
npx create-react-app dynamic-demo
cd dynamic-demo
npm start
```

5.13.4.2 *2. Folder Structure*

```
dynamic-demo/
 src/
     App.js
```

```
    components/
        Issue.js
```

5.13.4.3 *3. Create the Issue Component*

Listing 5.69: src/components/Issue.js

```
import React from "react";

function Issue({ id, title, status }) {
  return (
    <div style={{
      border: "1px solid #ccc",
      borderRadius: "6px",
      padding: "8px",
      margin: "6px 0",
      background: status === "Closed" ? "#f8f8f8" : "#e6ffed"
    }}>
      <strong>{title}</strong> - <em>{status}</em>
    </div>
  );
}

export default Issue;
```

5.13.4.4 *4. Create the Parent Component (App.js)*

Listing 5.70: src/App.js

```
import React from "react";
import Issue from "./components/Issue";

function App() {
  const issues = [
    { id: 1, title: "Login Bug", status: "Open" },
    { id: 2, title: "API Timeout", status: "Closed" },
    { id: 3, title: "UI Overlap", status: "In Progress" }
  ];

  return (
    <div style={{ width: "80%", margin: "auto" }}>
      <h1 style={{ textAlign: "center", color: "teal" }}>Issue Tracker
        </h1>
```

```
      {issues.length > 0 ? (
        issues.map(issue => (
          <Issue key={issue.id} {...issue} />
        ))
      ) : (
        <p>No issues found!</p>
      )}
    </div>
  );
}

export default App;
```

Explanation:

- The parent component defines an array `issues`.
- The `.map()` function loops through the array and renders one `<Issue />` for each object.
- The spread operator (`...issue`) passes all fields as props.
- A unique `key` is added to each child for React's internal tracking.
- Conditional rendering shows a message if no issues are available.

5.13.5 Step 5: Run and Observe Output

When you start the app using `npm start`, your browser will display:

Issue Tracker Login Bug — Open API Timeout — Closed UI Overlap — In Progress

Each item is created dynamically from the `issues` array.

5.13.6 Conditional Rendering

React allows rendering parts of the UI conditionally. Example — show different colors or messages based on data:

```
{issue.status === "Closed"
  ? <p style={{color: "gray"}}>Issue closed</p>
  : <p style={{color: "green"}}>Issue active</p>}
```

Or display nothing if the array is empty:

```
{issues.length === 0 && <p>No issues available!</p>}
```

5.13.7 Understanding the `key` Prop

The `key` prop helps React identify which elements changed. Keys should be:

- **Unique** among siblings (usually `id` or index).
- **Stable** — avoid using random numbers.

Example:

```
{students.map(student => (
  <Student key={student.rollNo} {...student} />
))}
```

5.13.8 Exercise: Render a Student List Dynamically

Goal: Practice rendering multiple components using array data.
Instructions:

1. Create a new React component `Student.js` inside the `components/` folder.
2. Use an array of student objects in `App.js`.
3. Render a list of `<Student />` components dynamically using `.map()`.

5.13.8.1 *Example Solution*

Student.js:

Listing 5.71: src/components/Student.js

```
import React from "react";

function Student({ rollNo, name, course, grade }) {
  return (
    <div style={{
      border: "1px solid #aaa",
      padding: "8px",
      borderRadius: "6px",
      margin: "5px 0",
      background: "#f0faff"
    }}>
      <strong>{name}</strong> ({rollNo})
      <p>Course: {course} - Grade: {grade}</p>
    </div>
  );
}

export default Student;
```

App.js:

Listing 5.72: src/App.js

```
import React from "react";
import Student from "./components/Student";

function App() {
  const students = [
    { rollNo: 101, name: "Ravi", course: "BCA", grade: "A" },
    { rollNo: 102, name: "Ananya", course: "BSc CS", grade: "A+" },
    { rollNo: 103, name: "Vikram", course: "B.Tech IT", grade: "B" }
  ];

  return (
    <div style={{ width: "70%", margin: "auto" }}>
      <h1 style={{ textAlign: "center" }}>Student List</h1>

      {students.map(student => (
        <Student key={student.rollNo} {...student} />
      ))}
    </div>
  );
}

export default App;
```

5.13.9 Output:

Student List Ravi (101) — Course: BCA — Grade: A Ananya (102) — Course: BSc CS — Grade: A+ Vikram (103) — Course: B.Tech IT — Grade: B

5.13.10 Tips for Dynamic Rendering

- Always include a unique `key` prop when using `.map()`.
- Use `conditional rendering` for empty states.
- You can combine `map()` with event handlers:

```
{issues.map(issue => (
  <button key={issue.id} onClick={() => alert(issue.title)}>
    {issue.title}
  </button>
))}
```

- Avoid mutating arrays directly — use React state (`useState`) when dynamically updating lists.

5.13.11 Summary

- Dynamic composition means rendering multiple components automatically from arrays.
- The `.map()` function converts data into JSX.
- The `key` prop helps React track list items efficiently.
- Conditional rendering allows showing or hiding parts of the UI.
- These techniques are essential for building lists, tables, dashboards, and feeds in real applications.

Key Takeaway: Dynamic rendering is one of React's core strengths — it turns simple data arrays into powerful, interactive UIs with minimal code.

CHAPTER 6

6 React State and API Integration Concepts

Modern web applications are not just static pages; they are interactive, data-driven systems that respond instantly to user actions and external data sources. React, one of the most popular JavaScript libraries for building user interfaces, achieves this through the concept of **state management**. State determines how a component behaves and what it displays at any given moment. Whenever state changes, React automatically re-renders the component tree to reflect the latest data. This makes React ideal for building dynamic interfaces that update in real time — such as forms, dashboards, or issue trackers.

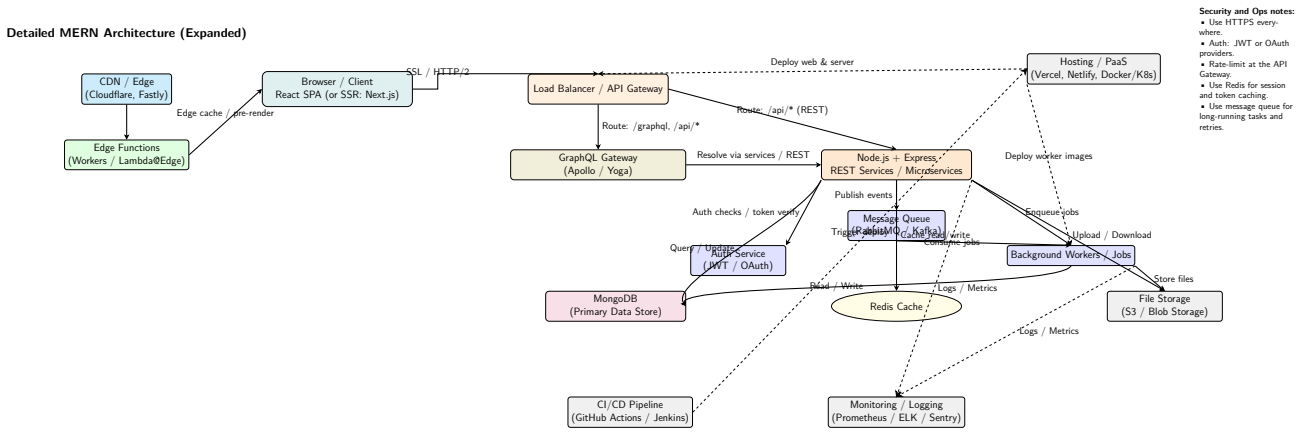


Figure 6.1: Expanded MERN architecture showing CDN/edge, client (SPA/SSR), API gateway, GraphQL gateway, Express services, auth/cache/queue, data stores (MongoDB, S3), background workers, and CI/CD / monitoring.

In large applications, multiple components often need to share or synchronize data. This introduces the concepts of **lifting state up**, **component hierarchy**, and **communication between components**. By structuring components properly and understanding the difference between **state** (data that can change) and **props** (data passed from parent to child), developers can design applications that are modular, maintainable, and predictable. Stateless and stateful components each play an important role: stateless components focus purely on presentation, while stateful components handle data, events, and logic.

Once the front-end logic is well-structured, it must connect to external data sources. This is where backend technologies like **Express.js**, **REST APIs**, and **GraphQL** come in. Express provides a lightweight framework for building web servers and APIs in Node.js, while REST defines a standard architectural style for exchanging data using HTTP methods. However, in complex systems where clients need flexible data queries, **GraphQL** provides a more powerful, strongly typed, and efficient alternative. It allows clients to request exactly the data they need from a single endpoint, minimizing network overhead and improving performance.

This unit bridges the gap between the **frontend and backend** in the MERN architecture. Students will learn how React components manage data internally using state and how that state can be synchronized

with data from APIs. They will also explore how to create, query, and integrate APIs using Express (for REST) and GraphQL (for advanced data operations). By the end of this unit, students will be capable of designing responsive, data-driven web applications where the React frontend seamlessly interacts with dynamic backend services.

6.1. React State and Component Communication

Modern user interfaces must dynamically respond to user interactions, data changes, and network updates. React provides a robust mechanism for handling such interactivity through **component state**. State determines how a component behaves and what it displays at a given time. Understanding how to define, initialize, and update state correctly is fundamental to building responsive applications.

6.1.1 Introduction to State Management

6.1.1.1 *Definition of State*

In React, **state** refers to data that can change over time and directly influences how a component is rendered. Every time state changes, React automatically re-renders the affected parts of the UI. State makes React components interactive and allows them to maintain data between renders.

Example: Counter Component

Listing 6.1: Simple counter using state

```
import React, { useState } from "react";

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

  return (
    <div style={{ textAlign: "center" }}>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}

export default Counter;
```

Each click updates the count state, causing React to re-render the component and display the new value.

6.1.1.2 *Difference Between State and Props*

Props and **state** both influence a component's output but serve different purposes:

- **Props** are read-only inputs passed from a parent component.
- **State** is internal and mutable — it belongs to the component itself.

Aspect	Props	State
Ownership	Passed from parent	Managed inside the component
Mutability	Immutable	Mutable via setters
Usage	Configure child component	Handle component data and behavior
Update trigger	Parent re-renders	Component re-renders itself

Example: Props vs. State

Listing 6.2: Props and state together

```
function Greeting({ initialName }) {   // prop from parent
  const [name, setName] = useState(initialName); // internal state

  return (
    <div>
      <h3>Hello, {name}!</h3>
      <button onClick={() => setName("React Learner")}>Change Name</
         button>
    </div>
  );
}
```

Here, `initialName` is fixed (prop), but `name` is managed locally (state).

6.1.1.3 *Importance of State in Dynamic UIs*

React applications rely on state for interactivity:

1. **User Interaction:** Buttons, inputs, and checkboxes change component behavior through state updates.
2. **Data-driven Rendering:** Lists and dashboards depend on data stored in state.
3. **Asynchronous Data:** API results or form submissions update the UI via state changes.
4. **Conditional UI:** Visibility toggles or themes are controlled using state flags.

Example: Theme Toggle

```
function ThemeSwitcher() {
  const [dark, setDark] = useState(false);
  return (
    <div style={{
      background: dark ? "#333" : "#fff",
      color: dark ? "#fff" : "#000",
      padding: 20
    }}>
      <p>Current theme: {dark ? "Dark" : "Light"}</p>
      <button onClick={() => setDark(!dark)}>Toggle Theme</button>
```

```
    </div>
  );
}
```

6.1.2 Initial State and State Initialization

6.1.2.1 *Setting Initial State using useState()*

The `useState()` hook initializes state. It returns an array with two elements:

1. The current state value.
2. A setter function to update the value.

Example: Initial Counter Value

```
const [count, setCount] = useState(5); // initial state = 5
```

Lazy Initialization: If calculating the initial state is expensive, pass a function:

```
const [data, setData] = useState(() => {
  console.log("Expensive computation only once");
  return computeInitialData();
});
```

6.1.2.2 *Async State Initialization*

Often, data comes from an API or database. In such cases, initialize state as empty and populate it asynchronously.

Example: Fetching API Data

Listing 6.3: Async initialization using useEffect

```
import React, { useState, useEffect } from "react";

function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    async function fetchUsers() {
      const res = await fetch("https://jsonplaceholder.typicode.com/
         users");
      const data = await res.json();
      setUsers(data);
      setLoading(false);
```

```
    }
    fetchUsers();
  }, []);

  if (loading) return <p>Loading users...</p>;

  return (
    <ul>
      {users.map(u => (
        <li key={u.id}>{u.name}</li>
      ))}
    </ul>
  );
}
```

Tip: Always initialize state with a value that matches the expected data type (e.g., an empty array for lists).

6.1.2.3 *Common Mistakes in Initial State Setup*

- **Mutating state directly:**

```
// Wrong
count = count + 1;

// Correct
setCount(count + 1);
```

- **Recomputing initial state on every render:**

```
//
const [data] = useState(expensiveCalculation());
//
const [data] = useState(() => expensiveCalculation());
```

- **Setting state during render:**

```
// Causes infinite loop
const [value, setValue] = useState(0);
setValue(5);
```

- **Ignoring prop updates:**

```
// Doesn't react to prop changes
const [val, setVal] = useState(props.initial);
```

```
  //  Sync using useEffect
  useEffect(() => setVal(props.initial), [props.initial]);
```

6.1.3 Updating State

6.1.3.1 *Using State Setter Functions*

The setter returned by `useState()` updates the state and triggers re-rendering.

Example: Simple Counter

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
    </>
  );
}
```

6.1.3.2 *Functional Updates and Previous State*

When new state depends on the old value, use the **functional update form**:

```
setCount(prevCount => prevCount + 1);
```

Example: Multiple Click Updates

```
<button onClick={() => {
  setCount(prev => prev + 1);
  setCount(prev => prev + 1);
}}>
  +2
</button>
```

Both increments will execute correctly because React batches updates using the functional form.

6.1.3.3 *Batching and Asynchronous State Updates*

React may combine several state updates into one re-render for efficiency. This is called **batching**. Because of this, reading the state immediately after setting it may show the old value.

```
setCount(count + 1);
console.log(count); // old value
```

To ensure accuracy:

- Use functional updates.
- Place dependent logic inside `useEffect()` when reacting to changes.

Example: Reacting to State Change

```
useEffect(() => {
  console.log("Count updated:", count);
}, [count]);
```

6.1.3.4 *Immutable State Patterns*

React state must be treated as **immutable**. Instead of modifying existing arrays or objects, create and set new copies.

Example: Updating a Todo List

```
function TodoList() {
  const [todos, setTodos] = useState([{ id: 1, text: "Learn React",
     done: false }]);

  const toggleTodo = (id) => {
    setTodos(prev =>
      prev.map(t => t.id === id ? { ...t, done: !t.done } : t)
    );
  };

  return (
    <ul>
      {todos.map(t => (
        <li key={t.id}>
          <label>
            <input type="checkbox" checked={t.done} onChange={() =>
               toggleTodo(t.id)} />
            {t.text}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

Best Practices:

- Never mutate arrays/objects directly (use spread syntax or `map()`).
- Treat state as read-only.
- Store only necessary data — derive computed values inside render.

Summary:

- State defines component data that changes over time.
- Initialize using `useState()` (simple or lazy form).
- Fetch remote data asynchronously in `useEffect()`.
- Always update state immutably using setter functions.
- Use functional updates to handle asynchronous batching safely.

6.2. Event Handling in React

React uses a consistent, cross-browser system for handling user interactions called the **Synthetic Event System**. Instead of attaching native DOM listeners directly, React wraps events into a unified object, improving performance and compatibility.

6.2.0.1 *Understanding Synthetic Events*

Synthetic events are React's wrappers around the browser's native events. They normalize event properties so that React works the same across all browsers. For example, `onClick`, `onChange`, and `onSubmit` behave identically on Chrome, Firefox, or Edge.

Key features:

- Consistent API across browsers.
- Pooled for performance (reused between events).
- Follow React's lifecycle and batching system.

Example: Synthetic Event Logging

```
function ClickLogger() {
  function handleClick(event) {
    console.log("Event type:", event.type);
    console.log("Event target:", event.target);
  }
  return <button onClick={handleClick}>Click Me</button>;
}
```

6.2.0.2 *Handling Events in Functional Components*

Event handlers in React are camelCase (e.g., `onClick`, `onChange`). You pass a function reference, not a function call.

Example: Button Click Counter

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

  function handleClick() {
    setCount(count + 1);
  }
```

```
  return (
    <div>
      <p>You clicked {count} times.</p>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
}
```

6.2.0.3 *Passing Parameters to Event Handlers*

You can pass parameters to event handlers using arrow functions or bind methods.

Listing 6.4: Passing arguments to handlers

```
function GreetingList() {
  const handleGreet = (name) => alert(`Hello, ${name}!`);

  return (
    <div>
      <button onClick={() => handleGreet("Ravi")}>Greet Ravi</button>
      <button onClick={() => handleGreet("Priya")}>Greet Priya</button>
    </div>
  );
}
```

6.2.0.4 *Form and Input Event Handling Examples*

React controls form elements via **controlled components**. Each input field stores its value in component state and updates via onChange events.

Example: Controlled Form Component

```
function SimpleForm() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Name: ${name}, Email: ${email}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input value={name} onChange={(e) => setName(e.target.value)}
           />
```

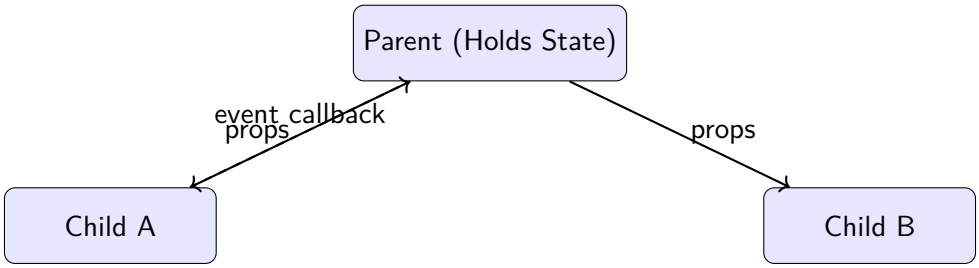
```
        </label><br/>
        <label>
          Email:
          <input value={email} onChange={(e) => setEmail(e.target.value)}
              />
        </label><br/>
        <button type="submit">Submit</button>
      </form>
    );
}
```

Concept: The input elements mirror the component's state, ensuring synchronization between UI and data.

6.2.1 Lifting State Up

6.2.1.1 *Data Flow Between Components*

React data flow is **unidirectional**: from parent to child through `props`. When two child components need to share the same data, that state is lifted to their common parent.



6.2.1.2 *When to Lift State*

Lift state up when:

- Two or more components need the same data.
- A parent must coordinate or synchronize behavior between children.
- Derived data depends on multiple child states.

6.2.1.3 *Practical Example: Shared Form Input*

Listing 6.5: Lifting state between two inputs

```
function TemperatureApp() {
  const [temperature, setTemperature] = useState("");

  return (
    <div>
      <TemperatureInput
        label="Celsius"
        value={temperature}
```

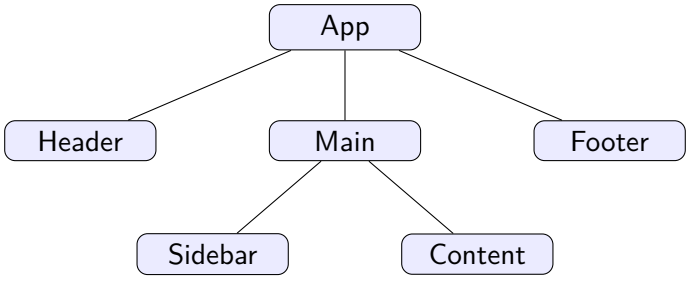
```
        onChange={setTemperature}
      />
      <TemperatureInput
        label="Fahrenheit"
        value={(temperature * 9/5) + 32}
        onChange={(v) => setTemperature((v - 32) * 5/9)}
      />
    </div>
  );
}

function TemperatureInput({ label, value, onChange }) {
  return (
    <div>
      <label>{label}: </label>
      <input
        value={value}
        onChange={(e) => onChange(Number(e.target.value))}
      />
    </div>
  );
}
```

Both input fields stay synchronized because they share state via the parent component.

6.2.2 Designing Components and Communication

6.2.2.1 *Component Hierarchy and Data Direction*

React applications are designed as trees of components. Data always flows **downward** from parent to child. Communication from child to parent happens through callback props.



6.2.2.2 *Parent-Child Communication Patterns*

- **Parent → Child:** via props.
- **Child → Parent:** via callback functions passed as props.
- **Sibling Sibling:** lift shared state to a common parent.

Example: Parent listening to child events

```
function Parent() {
  const handleSelect = (value) => alert(`Selected: ${value}`);
  return <Child onSelect={handleSelect} />;
}

function Child({ onSelect }) {
  return (
    <button onClick={() => onSelect("Option A")}>Choose</button>
  );
}
```

6.2.2.3 *Stateful vs. Stateless Components*

- **Stateful components** manage data and handle logic.
- **Stateless (presentational) components** only render UI based on props.

Example:

```
// Stateful container
function TodoContainer() {
  const [todos, setTodos] = useState(["Learn React", "Build App"]);
  return <TodoList items={todos} />;
}

// Stateless presentation
function TodoList({ items }) {
  return <ul>{items.map(t => <li key={t}>{t}</li>)}</ul>;
}
```

6.2.2.4 *Design Guidelines for React Components*

- Keep components small and focused.
- Lift state only when necessary.
- Use composition over inheritance.
- Avoid deep prop drilling — use context or state management when data is shared widely.
- Separate logic (containers) from presentation (UI).

6.2.3 State vs. Props ›

6.2.3.1 *Comparison Table and Use Cases*

Aspect	State	Props
Source	Managed within component	Passed from parent
Mutability	Mutable (via setter)	Immutable
Scope	Local to component	Accessible in child only
Use Case	Track changing data	Configure or initialize child
Change Trigger	User action or effect	Parent re-render

6.2.3.2 *When to Use State vs. Props*

- Use **state** when data changes over time (e.g., input fields, counters, toggles).
- Use **props** to pass data or behavior down (e.g., title, callback function).
- Convert state to props when sharing between components.

Example: Combining both

```
function Parent() {
  const [color, setColor] = useState("blue");
  return <Child color={color} onChange={setColor} />;
}

function Child({ color, onChange }) {
  return (
    <div style={{ color }}>
      <p>Selected color: {color}</p>
      <button onClick={() => onChange("red")}>Change to Red</button>
    </div>
  );
}
```

6.2.3.3 *Stateless Components and Pure UI*

Stateless components do not manage any internal state. They receive all data through props, making them predictable and easy to test.

Example:

```
function Welcome({ user }) {
  return <h2>Welcome, {user}!</h2>;
}
```

They are often called **pure components** when their output depends only on props and they do not cause side effects.

Summary:

- React uses synthetic events for consistent cross-browser event handling.
- State should be lifted up when shared by multiple components.
- Components communicate through props and callback functions.
- Stateless components handle only presentation; stateful ones manage data.

- Understand when to use state vs props for clean, maintainable architecture.

6.3. Express.js and REST API Fundamentals

The backend of a MERN application is typically powered by **Express.js**, a lightweight web framework for Node.js. It provides a simple, unopinionated way to build HTTP servers and REST APIs that can serve data to the frontend (React) or mobile clients.

6.3.1 Introduction to Express.js

6.3.1.1 *Overview and Role in MERN Stack*

Express.js is a Node.js framework used for building APIs and web applications. In the MERN architecture:

- React (Frontend) handles user interface and interactivity.
- Express (Backend) processes requests and responses.
- MongoDB stores data persistently.
- Node.js provides the runtime environment for Express.

Key features of Express:

- Simplifies creation of routes (URLs) and HTTP methods.
- Supports middleware for pre/post-processing of requests.
- Integrates easily with databases (e.g., MongoDB).
- Enables building RESTful APIs for web and mobile.

6.3.1.2 *Creating a Basic Express Server*

Step 1: Initialize Project

Listing 6.6: Creating an Express app

```
mkdir express-server
cd express-server
npm init -y
npm install express
```

Step 2: Create `server.js`

Listing 6.7: Minimal Express server

```
const express = require('express');
const app = express();
const PORT = 3000;

// Route definition
app.get('/', (req, res) => {
  res.send('Hello World from Express Server!');
```

```
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

Step 3: Run the server

```
node server.js
```

Visit http://localhost:3000 in the browser — you should see:

Hello World from Express Server!

6.3.1.3 *Middleware and Routing Concepts*

Middleware are functions that run before or after route handlers to process requests or modify responses. Common uses include:

- Logging and analytics.
- Authentication and authorization.
- Request body parsing (JSON, forms).
- Error handling.

Example: Middleware and route usage

```
const express = require('express');
const app = express();

// Example of middleware
app.use(express.json()); // built-in JSON parser
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // move to next middleware or route
});

// Example route
app.get('/api/hello', (req, res) => {
  res.json({ message: 'Hello from Middleware Demo!' });
});
```

6.3.2 REST API Design Principles

6.3.2.1 *What is a RESTful API?*

REST (Representational State Transfer) is an architectural style for designing APIs. A RESTful API uses standard HTTP methods to access and manipulate resources.

Core principles:

- Stateless — each request is independent.
- Uniform interface — consistent resource URLs and HTTP verbs.
- Layered system — can include caching, proxies, etc.
- Resource-based — identified using URLs like `/api/users`.

6.3.2.2 *HTTP Methods and CRUD Operations*

HTTP Method	CRUD Action	Example Endpoint
GET	Read	`/api/users` (Get all users)
POST	Create	`/api/users` (Add new user)
PUT	Update	`/api/users/:id` (Update existing user)
DELETE	Delete	`/api/users/:id` (Remove user)

6.3.2.3 *Endpoints, Parameters, and Status Codes*

Each API route is called an **endpoint**. Endpoints can include:

- **Path Parameters:** e.g., `/users/:id`
- **Query Parameters:** e.g., `/users?role=admin`

Common Status Codes:

- `200 OK` — Successful response
- `201 Created` — Resource successfully created
- `400 Bad Request` — Invalid input
- `404 Not Found` — Resource not found
- `500 Internal Server Error` — Server-side error

6.3.3 Building a REST API with Express ›

6.3.3.1 *Setting Up Routes (GET, POST, PUT, DELETE)*

Example: `users.js` — RESTful Routes

```
const express = require('express');
const app = express();
app.use(express.json());

let users = [
  { id: 1, name: 'Ravi', email: 'ravi@mail.com' },
  { id: 2, name: 'Priya', email: 'priya@mail.com' }
];

// GET all users
```

```
app.get('/api/users', (req, res) => res.json(users));

// GET single user
app.get('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id === Number(req.params.id));
  user ? res.json(user) : res.status(404).json({ error: 'User not found
     ' });
});

// POST create new user
app.post('/api/users', (req, res) => {
  const newUser = { id: Date.now(), ...req.body };
  users.push(newUser);
  res.status(201).json(newUser);
});

// PUT update user
app.put('/api/users/:id', (req, res) => {
  const id = Number(req.params.id);
  users = users.map(u => (u.id === id ? { ...u, ...req.body } : u));
  res.json({ message: 'User updated' });
});

// DELETE user
app.delete('/api/users/:id', (req, res) => {
  const id = Number(req.params.id);
  users = users.filter(u => u.id !== id);
  res.json({ message: 'User deleted' });
});

app.listen(4000, () => console.log('API running on http://localhost
   :4000'));
```

6.3.3.2 *Request and Response Objects*

Every route callback receives two main arguments:

- **req (Request)** — Contains details about the incoming HTTP request (URL, parameters, headers, body).
- **res (Response)** — Used to send the reply (text, JSON, status code) to the client.

Example: Accessing request data

```
app.post('/api/register', (req, res) => {
  console.log(req.body); // JSON data sent by client
  res.status(201).json({ message: 'User registered' });
});
```

6.3.3.3 *JSON Response Formatting*

Most modern APIs use JSON for data exchange. In Express, you can send JSON easily using `res.json()`.

```
app.get('/api/info', (req, res) => {
  res.json({
    app: 'MERN Example',
    version: '1.0.0',
    timestamp: new Date()
  });
});
```

6.3.3.4 *Testing APIs using Postman*

Postman is a popular tool to test REST APIs manually.

Steps:

1. Open Postman and enter http://localhost:4000/api/users.
2. Choose method: GET, POST, PUT, or DELETE.
3. For POST/PUT — go to "Body → raw → JSON" and add test data.
4. Press "Send" to view the response and status code.

6.3.4 Connecting REST API to Frontend (Preview) ›

The React frontend communicates with the Express API through HTTP requests.

6.3.4.1 *CORS and Middleware Setup*

By default, browsers block cross-origin requests. Enable **CORS (Cross-Origin Resource Sharing)** in Express:

```
npm install cors
```

```
const cors = require('cors');
app.use(cors()); // allow frontend to access API
```

6.3.4.2 *Fetch and Axios Integration*

Example: Using Fetch in React

```
useEffect(() => {
  fetch('http://localhost:4000/api/users')
    .then(res => res.json())
    .then(data => setUsers(data));
}, []);
```

Example: Using Axios

```
import axios from 'axios';

useEffect(() => {
  axios.get('http://localhost:4000/api/users')
       .then(res => setUsers(res.data))
       .catch(console.error);
}, []);
```

6.3.4.3 *Dynamic Rendering of Fetched Data*

Example: Displaying User List in React

```
function UserList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch('http://localhost:4000/api/users')
      .then(res => res.json())
      .then(data => setUsers(data));
  }, []);

  return (
    <ul>
      {users.map(u => (
        <li key={u.id}>{u.name} ({u.email})</li>
      ))}
    </ul>
  );
}
```

This demonstrates the full cycle:

- React sends a request to the Express API.
- Express retrieves and returns data as JSON.
- React dynamically renders it into the UI.

Summary:

- Express.js is the backend engine of the MERN stack.
- REST APIs follow CRUD conventions via HTTP methods.
- Middleware enhances Express by adding preprocessing logic.
- Postman is essential for testing API endpoints.
- Frontend integration (React + Fetch/Axios) completes the MERN communication loop.

6.4. Introduction to GraphQL

GraphQL is a query language and runtime for APIs developed by Facebook in 2012 and open-sourced in 2015. It provides a more flexible, efficient, and strongly typed alternative to REST APIs. While REST exposes multiple endpoints for different data needs, GraphQL allows clients to query exactly the data they require—no more, no less—through a single unified endpoint.

6.4.1 Understanding GraphQL vs REST

6.4.1.1 *Limitations of REST*

Traditional REST APIs follow a resource-based approach: each endpoint corresponds to a particular data entity or action. Although simple, this approach introduces several inefficiencies when applications grow in complexity.

Common REST limitations:

- **Over-fetching:** Clients receive more data than needed. Example: Fetching a user also returns their full address and profile image when only their name is needed.
- **Under-fetching:** Clients receive too little data, requiring multiple requests to complete one view. Example: Fetch user details, then make a second request for their posts.
- **Multiple Endpoints:** REST often requires several endpoints (`/users`, `/posts`, `/comments`) for related data.
- **Versioning Problems:** Updating APIs may require new versions (`/v1/users`, `/v2/users`) that increase maintenance overhead.

Example: REST requests to display a user profile

```
GET /api/users/101
GET /api/users/101/posts
GET /api/users/101/comments
```

The client must call multiple endpoints to gather data for a single page.

6.4.1.2 *Graph-Based Query Model*

GraphQL solves REST's inefficiencies by organizing data as a **graph of connected entities**. Clients can query related data (like users, posts, and comments) in a single request using a declarative syntax.

Example: GraphQL Query

Listing 6.8: A single GraphQL query fetching related entities

```
{
  user(id: 101) {
    name
    email
    posts {
      title
      comments {
```

```
        text
        author {
          name
        }
      }
    }
  }
}
```

Response:

```
{
  "data" {
    "user" {
      "name" "Ravi",
      "email" "ravi@example.com",
      "posts" [
        {
          "title" "Learning GraphQL",
          "comments" [
            { "text" "Great post!", "author": { "name": "Priya" } }
          ]
        }
      ]
    }
  }
}
```

Key advantage: Only one request fetches all necessary nested data. This approach reduces network calls and improves performance in large, data-driven applications.

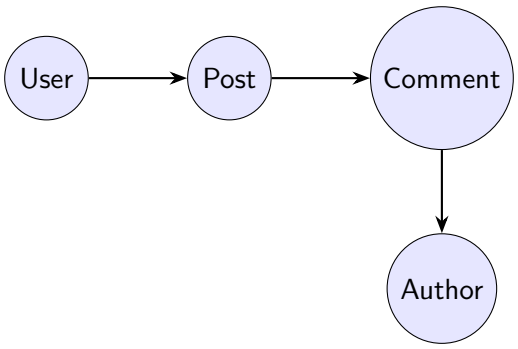


Figure: GraphQL treats data as a connected graph, allowing nested queries in one call.

6.4.1.3 *Single Endpoint Architecture*

Unlike REST, which exposes multiple endpoints for different resources, GraphQL serves all queries through a single endpoint, typically:

```
POST /graphql
```

Advantages:

- One consistent API URL — simpler for clients.
- Reduces redundant endpoints.
- Server and client communicate using structured query syntax (GraphQL documents).

Example: Express + Apollo Server Setup

Listing 6.9: Single GraphQL endpoint using Apollo Server

```
const { ApolloServer, gql } = require('apollo-server-express');
const express = require('express');
const app = express();

// Define schema
const typeDefs = gql`
  type User {
    id: ID!
    name: String!
    email: String!
  }
  type Query {
    users: [User]
  }
`;

// Define resolvers
const resolvers = {
  Query: {
    users: () => [
      { id: 1, name: "Ravi", email: "ravi@mail.com" },
      { id: 2, name: "Priya", email: "priya@mail.com" }
    ]
  }
};

// Create and apply Apollo server
const server = new ApolloServer({ typeDefs, resolvers });
await server.start();
server.applyMiddleware({ app, path: "/graphql" });

app.listen(4000, () => console.log("GraphQL running on http://localhost
   :4000/graphql"));
```

Now clients can send all queries (for users, posts, etc.) to the same endpoint:

```
POST http://localhost:4000/graphql
```

6.4.1.4 *Strongly Typed Schema System*

GraphQL APIs are defined using a **Schema Definition Language (SDL)** that describes the exact shape of data and relationships.

Example Schema:

Listing 6.10: GraphQL type definitions

```
type User {
  id: ID!
  name: String!
  email: String!
  age: Int
}

type Query {
  users: [User]
  user(id: ID!): User
}
```

Explanation:

- `type User` defines a data model with fields and their types.
- `Query` defines read operations (similar to GET in REST).
- The exclamation mark (`!`) means the field is required.

Benefits:

- Prevents invalid queries at compile time.
- Provides clear contract between client and server.
- Enables strong IDE support and autocompletion.

6.4.1.5 *Introspection and Self-Documentation*

One of GraphQL's unique strengths is its built-in **introspection system**. It allows clients (and developers) to query the API for its schema definition — meaning every GraphQL API can describe itself.

Example Introspection Query:

```
{
  __schema {
    types {
      name
      fields { name }
    }
  }
}
```

Result:

```
{
  "data" {
    "__schema" {
      "types" [
        {
          "name" "User",
          "fields" [ { "name": "id" }, { "name": "name" }, { "name": "
              email" } ]
        }
      ]
    }
  }
}
```

Developer Tools:

- **GraphiQL** — an in-browser IDE for running queries.
- **Apollo Sandbox** — interactive GraphQL query console.
- **VS Code Plugins** — autocomplete and schema validation tools.

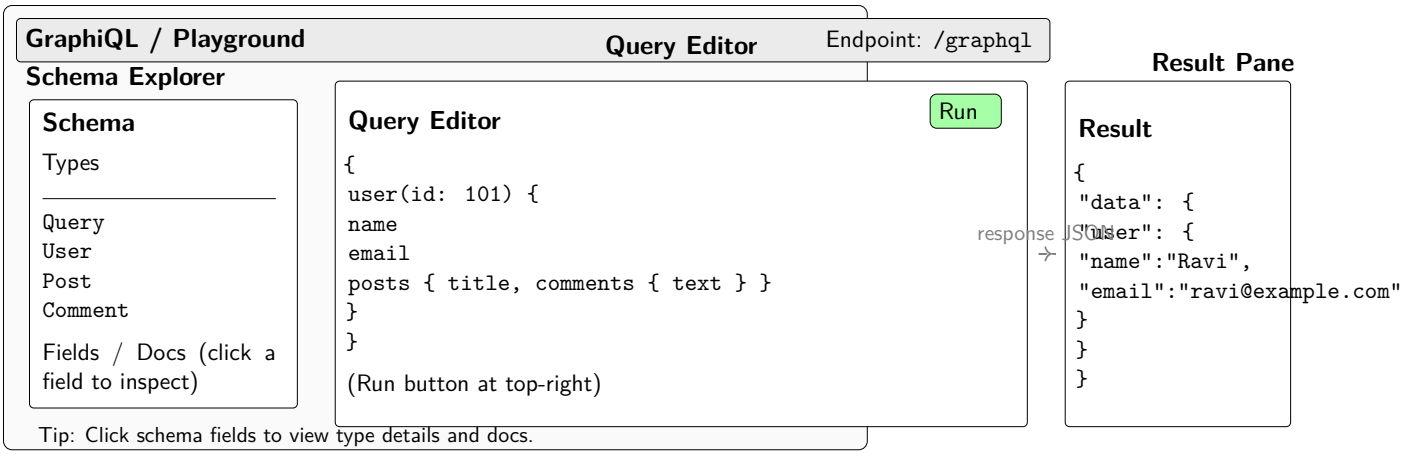


Figure 6.2: GraphiQL / Playground layout: left — schema explorer; centre — query editor; right — result pane.

Summary:

- GraphQL replaces multiple REST endpoints with one flexible query endpoint.
- It prevents over-fetching and under-fetching by allowing clients to request exactly what they need.
- GraphQL's schema is strongly typed and introspectable, making APIs self-documenting.
- Ideal for modern applications requiring efficient data fetching and multiple data relationships.

CHAPTER 7

7 MongoDB

7.1. Learning Objectives

In this chapter, students will develop a strong foundation in the concepts and practical applications of MongoDB, a popular NoSQL database widely used in modern web development. The primary goal of this chapter is to equip learners with the ability to design, implement, and interact with databases using MongoDB, both from the command line and through programmatic access via Node.js.

Students will begin by exploring the core architecture of MongoDB, including the document-oriented data model and the structure of collections and databases. They will gain a clear understanding of how MongoDB differs from traditional relational databases and why it has become an essential part of full-stack web development, especially within the MERN (MongoDB, Express, React, Node.js) ecosystem.

Through guided exercises, students will learn how to perform CRUD operations—Create, Read, Update, and Delete—using both the Mongo Shell and the MongoDB Node.js driver. They will also explore the aggregation framework, which enables powerful data processing and transformation within the database. By mastering these fundamental skills, learners will be able to efficiently manage and query large, complex data sets.

Beyond basic database operations, this chapter will introduce students to building RESTful APIs using Node.js and Express, providing a bridge between the database and client-side applications. This integration forms the backbone of modern web systems, allowing data to be dynamically retrieved, updated, and displayed to users.

In the later sections, students will gain insight into front-end interaction with MongoDB-powered back-ends using React. They will learn how to modularize and bundle client-side projects with Webpack, ensuring efficient organization and optimized performance. The chapter will conclude with practical discussions on debugging and deploying full-stack applications, emphasizing the workflow from local development to production deployment.

By the end of this chapter, students will have developed both theoretical understanding and practical competence in using MongoDB within the broader context of full-stack web development. They will be able to confidently design data models, connect applications to databases, and deploy complete, modular web systems powered by MongoDB.

7.2. Introduction to MongoDB

7.2.1 Objectives

This section introduces the learner to the foundational ideas behind MongoDB and the NoSQL paradigm. The objective is to develop an intuitive and practical understanding of MongoDB's architecture, its document-

oriented data model, and the scenarios in which a NoSQL approach is preferable to a relational one. After studying this section the student should be able to explain how MongoDB stores and organizes data, how documents relate to collections and databases, and the role BSON plays in internal representation. The student will also be able to install MongoDB (locally or in the cloud via Atlas), start a shell session, and perform basic data creation and retrieval tasks.

7.2.2 Topics — detailed explanation

Introduction to MongoDB and its advantages. MongoDB is a modern, document-oriented database designed for ease of development and horizontal scalability. Unlike traditional relational databases, MongoDB stores data in flexible, self-describing documents (similar to JSON) rather than in rigid tables with fixed schemas. This flexibility makes it straightforward to evolve an application's data model as requirements change: fields can be added to some documents without requiring costly schema migration operations. MongoDB also provides built-in replication and sharding features that make it relatively simple to scale reads and writes across multiple machines. The combination of an expressive query language, secondary indexes, and an aggregation framework gives developers powerful tools for querying and transforming data while keeping the operational model simple.

Databases, Collections, and Documents. At the core of MongoDB's model are three nested concepts: databases, collections, and documents. A *database* is a namespace that holds a set of collections and the related metadata. Within a database, a *collection* is a grouping of documents that are typically related by a common purpose (for example, a 'users' collection or an 'orders' collection). A *document* is the basic unit of data — an ordered set of key-value pairs that can contain nested documents and arrays. Documents are highly expressive: they can store complex, hierarchical data in a single record, which often eliminates the need for the multi-table joins commonly used in relational systems. This design maps closely to how application-level objects are modeled in code, reducing the impedance mismatch between application and storage.

JSON vs BSON data representation. MongoDB documents are represented in a JSON-like format for human readability, but internally they are stored in BSON (Binary JSON). BSON preserves the flexibility and readability of JSON while adding type information and efficient binary encodings for common data types (such as dates and binary data). BSON's binary format enables fast parsing and compact storage while allowing MongoDB to store additional data types (for example, 32-bit and 64-bit integers, floating point types, and a native BSON date type) that are not part of standard JSON. When working interactively in the shell or when exchanging data over APIs, documents are often shown as JSON for clarity, but developers should be aware of BSON's role in performance and storage behavior.

SQL vs NoSQL comparison. Relational (SQL) and document (NoSQL) databases take different approaches to modeling and accessing data. SQL databases center on normalized tables, strong schemas, and multi-table joins. They excel at complex transactions with strict consistency requirements and when the data model is stable. NoSQL document databases, by contrast, emphasize schema flexibility, denormalization, and horizontal scalability. MongoDB is particularly well suited for use cases where data structures evolve frequently, where application objects map naturally to hierarchical documents, or where you need to scale out across many commodity servers. That said, MongoDB also supports transactions and strong consistency

guarantees for many workloads, so the choice between SQL and NoSQL is a trade-off that depends on the application's functional and non-functional requirements rather than a strict technical limitation.

Installation and MongoDB Atlas setup. Getting started with MongoDB can be done in two common ways: installing a local server for development, or using MongoDB Atlas, the hosted cloud service. A local installation is useful for offline development and learning: official installers are available for Windows, macOS, and major Linux distributions, and the installation typically provides the 'mongod' server process and the interactive 'mongosh' shell. MongoDB Atlas offers a managed cluster in the cloud with a free-tier option, automated backups, monitoring dashboards, and simplified clustering and security configuration. Atlas is especially convenient for teaching and prototyping since it removes much of the operational overhead; the instructor can provide connection URIs for students to use from their development machines. In either case, students will learn how to configure basic authentication, create an initial database, and connect to the server using the shell or client drivers.

7.2.3 Hands-on — Expanded Practical Exercise ›

Practical Exercise: First Steps with MongoDB

1. **Install or provision MongoDB.** Choose either a local installation of MongoDB Community Server or create a free cluster in MongoDB Atlas. For a local install, follow the platform-specific instructions to install the server and the MongoDB Shell (`mongosh`). For Atlas, sign up, create a free-tier cluster, and configure IP access and a database user.

2. **Connect using the shell.** Start the MongoDB server (if local) and open `mongosh`. For Atlas, copy the connection string from the Atlas console and connect from your shell or a client using the provided URI. Verify the connection by running `db.version()` and observing the server response.

3. **Create a database and collection.** In the shell, switch to a new database with:

   ```
   use mycourse
   ```

   This command creates the namespace only after you insert data. Create a collection implicitly by inserting a document:

   ```
   db.students.insertOne({ name: "Aisha", enrolled: true, scores: [85, 92] })
   ```

   Confirm the collection exists using:

   ```
   show collections
   ```

   Inspect the inserted document:

   ```
   db.students.find().pretty()
   ```

4. **Insert and query sample documents.** Add several documents that demonstrate nested fields and arrays. Practice queries that retrieve documents by simple equality, by range, and by matching fields within embedded documents or arrays. For example:

```
db.students.insertMany([
  { name: "Ravi", enrolled: true, profile: { year: 2 } },
  { name: "Leela", enrolled: false, profile: { year: 1 } }
])

db.students.find({ "profile.year": 2 })
db.students.find({ enrolled: true }).projection({ name: 1 })
```

5. **Reflect and record.** After completing the exercises, write a short paragraph summarizing the differences you observed between inserting and querying data in MongoDB compared to relational databases. Reflect on the flexibility of the schema and the ease of working with nested data.

7.3. Mongo Shell and CRUD Operations

7.3.1 Objectives

This section focuses on performing the fundamental database operations known as CRUD—Create, Read, Update, and Delete—using the MongoDB Shell. The objective is to provide learners with both conceptual understanding and practical experience in manipulating data within MongoDB. By the end of this section, students will be able to insert, retrieve, modify, and remove documents from collections, as well as apply the aggregation framework for more advanced data analysis.

7.3.2 Detailed Discussion

CRUD operations form the backbone of any database interaction. In MongoDB, these operations are performed using simple yet powerful commands that interact with documents within collections. The Mongo Shell (`mongosh`) provides an interactive environment to experiment with these commands directly.

Create Operations. The process of adding new data to MongoDB collections is known as creating documents. This can be done using the `insertOne()` and `insertMany()` methods. The `insertOne()` command adds a single document, while `insertMany()` allows multiple records to be inserted in a single operation. Each document is stored as a JSON-like structure, automatically assigned a unique `_id` field by MongoDB unless explicitly provided by the user. The flexibility of MongoDB's document structure allows different documents in the same collection to have varying fields, making it ideal for dynamic or evolving data models.

Read Operations. Reading data from a collection is achieved with the `find()` and `findOne()` commands. The `find()` method returns all matching documents that meet the specified criteria, while `findOne()` retrieves only the first matching document. Queries are expressed as JSON objects, allowing rich filtering using comparison, logical, and element operators. For example, a query such as:

```
db.users.find({ age: { $gt: 25 } })
```

returns all documents where the age field is greater than 25. The `find()` method can also take an optional projection parameter that limits which fields are returned, helping optimize data retrieval and reduce clutter.

Projection. Projection allows the developer to control the visibility of fields in query results. For instance, to display only the `name` and `email` fields while excluding the `_id`, one can write:

```
db.users.find({}, { name: 1, email: 1, _id: 0 })
```

This feature is particularly useful when only certain attributes are required for display or further processing.

Update Operations. MongoDB provides the `updateOne()` and `updateMany()` methods to modify existing documents. Updates use operators such as `$set` to assign new values, `$inc` to increment numerical fields, and `$push` to append elements to arrays. For example, to increase the score of all students by five points:

```
db.students.updateMany({}, { $inc: { score: 5 } })
```

Updates can be targeted at specific documents using filters, ensuring precise data manipulation.

Delete Operations. The `deleteOne()` and `deleteMany()` methods remove documents from a collection. As their names suggest, the former deletes only the first matching document, while the latter removes all documents that satisfy the filter. Developers must exercise caution with delete operations, as they are irreversible.

Aggregation Framework. Beyond basic queries, MongoDB provides an aggregation framework that allows sophisticated data analysis and transformation. Using a pipeline structure, developers can filter, group, sort, and reshape data with great efficiency. Common stages include:

- `$match` — filters documents based on specific conditions.
- `$group` — groups documents by a key and performs operations such as sum or average.
- `$sort` — arranges the output documents in a defined order.
- `$project` — reshapes each document by including, excluding, or computing new fields.

For example, to compute the average age of users grouped by gender:

```
db.users.aggregate([
  { $group: { _id: "$gender", avgAge: { $avg: "$age" } } }
])
```

Through these CRUD and aggregation operations, students gain a complete understanding of data management in MongoDB and its expressive query capabilities.

7.3.3 Hands-on — Practical Exercise ›

Practical Exercise: CRUD Operations in the Mongo Shell

1. **Create a collection.** In the shell, create a new collection named `users` and insert a few sample documents:

```
db.users.insertMany([
  { name: "Anita", age: 24, city: "Pune" },
  { name: "Ramesh", age: 30, city: "Delhi" },
  { name: "Maya", age: 28, city: "Chennai" }
])
```

2. **Read data.** Retrieve all documents using:

```
db.users.find().pretty()
```

   Then, filter results to find users above 25 years:

```
db.users.find({ age: { $gt: 25 } })
```

3. **Project selected fields.** Display only user names and cities:

```
db.users.find({}, { name: 1, city: 1, _id: 0 })
```

4. **Update records.** Increase all users' ages by one year:

```
db.users.updateMany({}, { $inc: { age: 1 } })
```

5. **Delete a document.** Remove one user whose name is "Ramesh":

```
db.users.deleteOne({ name: "Ramesh" })
```

6. **Aggregation practice.** Group remaining users by city and count them:

```
db.users.aggregate([
  { $group: { _id: "$city", totalUsers: { $sum: 1 } } }
])
```

7. **Reflection.** Record your observations on how MongoDB's document-oriented operations differ from SQL statements such as `INSERT`, `SELECT`, `UPDATE`, and `DELETE`. Consider the flexibility of documents and the use of operators within MongoDB queries.

7.4. MongoDB with Node.js and Mongoose

7.4.1 Objectives

In this section, students will learn how to integrate Node.js applications with MongoDB using both the official MongoDB driver and the popular Mongoose library. The goal is to demonstrate two common approaches to interacting with a MongoDB database: direct access through the native driver and abstraction through an ODM (Object Data Modeling) layer. Students will gain hands-on experience performing CRUD operations, defining schemas and models, and handling asynchronous operations and errors effectively using `async/await`.

7.4.2 Detailed Discussion

From the Native Driver to Mongoose. The MongoDB Node.js driver provides low-level access to MongoDB databases and collections, allowing developers to interact directly with documents. While it offers full control and flexibility, it requires developers to handle validation, schema consistency, and relationship management manually.

To simplify these tasks, the **Mongoose** library acts as an Object Data Modeling (ODM) tool. It provides a schema-based abstraction on top of MongoDB, enabling developers to define data structures, apply validation rules, and manage relationships between collections in a structured and maintainable way. Mongoose also integrates seamlessly with JavaScript's asynchronous patterns, making it a preferred choice for large-scale applications.

Installing and Setting Up Mongoose. To get started, install both MongoDB and Mongoose packages using npm:

```
npm install mongodb mongoose
```

Import the Mongoose module into your project and establish a connection:

```
const mongoose = require('mongoose');

async function connectDB() {
  try {
    await mongoose.connect('mongodb://localhost:27017/school');
    console.log('Connected to MongoDB via Mongoose');
  } catch (err) {
    console.error('Connection error:', err);
  }
}

connectDB();
```

When the `connect()` function is called, Mongoose establishes a connection pool to the specified MongoDB instance, allowing multiple operations to be executed concurrently and efficiently.

Defining Schemas and Models. In Mongoose, data structure and validation rules are defined through *schemas*. A schema defines the shape of documents within a collection, including field types, default values, and constraints. A model, in turn, is a compiled version of the schema that represents a collection in the database and provides an interface for CRUD operations.

Example schema and model:

```
const studentSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: { type: Number, min: 0 },
  enrolled: { type: Boolean, default: true },
  marks: { type: Number, default: 0 }
});

const Student = mongoose.model('Student', studentSchema);
```

Here, the `Student` model maps to a `students` collection in the database. The schema enforces rules such as requiring the `name` field and ensuring that `age` is a non-negative number.

CRUD Operations with Mongoose. Mongoose simplifies CRUD operations with intuitive methods on models.

Create:

```
await Student.create({ name: "Aisha", age: 21, marks: 88 });
```

Read:

```
const students = await Student.find({ enrolled: true });
console.log(students);
```

Update:

```
await Student.updateOne({ name: "Aisha" }, { marks: 90 });
```

Delete:

```
await Student.deleteOne({ name: "Aisha" });
```

Each of these operations returns a Promise, enabling easy integration with `async/await` for clean and readable asynchronous code.

Schema Validation and Middleware. Mongoose provides built-in validation and middleware (also called hooks) that execute before or after database operations. Validation ensures data consistency, while middleware allows developers to automate tasks like logging or data transformation. For example:

```
studentSchema.pre('save', function(next) {
  console.log(`About to save: ${this.name}`);
  next();
});
```

This `pre-save` hook executes automatically whenever a student document is saved.

Error Handling and Async/Await. Mongoose operations return Promises, so all database operations should be wrapped in `try...catch` blocks for error handling. For example:

```
try {
  const newStudent = await Student.create({ name: "Raj", age: -5 });
} catch (err) {
  console.error("Validation error:", err.message);
}
```

In this example, Mongoose throws a validation error because the age does not meet the schema requirements. Such validation at the application layer prevents invalid or inconsistent data from being stored in the database.

When to Use Mongoose. While the MongoDB native driver is suitable for simple, small-scale applications or utility scripts, Mongoose excels in projects that require:

- Structured and consistent data models.
- Built-in validation and middleware logic.
- Readable, maintainable codebases.
- Easier integration with REST APIs and MVC architectures.

7.4.3 Hands-on — Practical Exercise

Practical Exercise: Using Mongoose for Database Operations

1. **Initialize a Node.js project.** Create a new project directory, initialize npm, and install dependencies:

   ```
   mkdir mongoose-demo
   cd mongoose-demo
   npm init -y
   npm install mongoose
   ```

2. **Connect to MongoDB.** Create a file named `app.js` and connect to MongoDB using Mongoose:

   ```
   const mongoose = require('mongoose');
   mongoose.connect('mongodb://localhost:27017/school')
     .then(() => console.log('Connected successfully'))
     .catch(err => console.error(err));
   ```

3. **Define a schema and model.** Add a student schema to define structure and constraints:

   ```
   const studentSchema = new mongoose.Schema({
     name: String,
     age: Number,
     course: String
   });
   ```

```
const Student = mongoose.model('Student', studentSchema);
```

4. **Perform CRUD operations.** Insert a document and display all records:

```
await Student.create({ name: "Meera", age: 22, course: "Node.js" });
const students = await Student.find();
console.log(students);
```

Update a record:

```
await Student.updateOne({ name: "Meera" }, { course: "React" });
```

Delete a record:

```
await Student.deleteOne({ name: "Meera" });
```

5. **Experiment with validation.** Add validation rules (e.g., require `name`) and try saving invalid documents to observe Mongoose's error-handling behavior.
6. **Reflection.** Compare the experience of using Mongoose with using the native MongoDB driver. Note how Mongoose simplifies schema management, validation, and query syntax.

7.5. Express.js and RESTful APIs

7.5.1 Objectives

In this module, students will learn how to develop RESTful APIs using the Express.js framework in combination with MongoDB. The goal is to understand how server-side applications manage data flow between client and database, handle HTTP requests and responses, and apply middleware for validation and error handling. By the end of this section, learners will be able to create, read, update, and delete data from MongoDB through REST endpoints, forming a key component of the MERN (MongoDB, Express, React, Node.js) stack.

7.5.2 Detailed Discussion

Introduction to Express.js. Express.js is a lightweight and flexible web application framework built on top of Node.js. It simplifies the process of creating web servers by providing a robust set of tools for handling routes, middleware, and HTTP requests. Express abstracts much of the repetitive boilerplate code required by the native Node.js HTTP module, allowing developers to focus on application logic instead of low-level

networking details.

Express applications follow a modular structure composed of routes, controllers, and middleware. This structure promotes clarity, reusability, and maintainability, making Express the standard choice for backend development in Node.js ecosystems.

Understanding REST Architecture. REST (Representational State Transfer) is an architectural style that defines how clients and servers communicate over HTTP. In RESTful APIs, data is represented as resources, each accessible through a unique URL. Each HTTP method corresponds to a specific type of operation:

- **GET** – Retrieve data from the server.
- **POST** – Send new data to the server.
- **PUT** – Update existing data.
- **DELETE** – Remove data.

For example, in a student management API:

- `GET /students` retrieves all students.
- `GET /students/:id` retrieves a specific student.
- `POST /students` adds a new student.
- `PUT /students/:id` updates a student's record.
- `DELETE /students/:id` removes a student.

These conventions ensure that REST APIs are consistent and predictable for clients.

Setting Up an Express Server. To create a new Express server, install the Express package and initialize the application:

```
npm install express mongoose
```

A minimal Express server is shown below:

```
const express = require('express');
const app = express();
app.use(express.json()); // Parse JSON bodies

app.get('/', (req, res) => {
  res.send('Welcome to the Express API!');
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

The `express.json()` middleware enables JSON parsing, allowing the server to handle incoming request bodies sent in JSON format.

Connecting Express to MongoDB via Mongoose. To connect Express routes with MongoDB, use Mongoose to define models and perform CRUD operations. After establishing a connection (as covered in the previous module), import the model into your route handlers.

Example connection setup:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/school')
  .then(() => console.log('Database connected'))
  .catch(err => console.error('Connection error', err));
```

Creating RESTful Routes. With Express, routes define how the server responds to various HTTP methods. Here is an example of REST routes for a `Student` resource:

```
const Student = require('./models/student'); // import the Mongoose model

// Create a new student
app.post('/students', async (req, res) => {
  try {
    const student = await Student.create(req.body);
    res.status(201).json(student);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// Retrieve all students
app.get('/students', async (req, res) => {
  const students = await Student.find();
  res.json(students);
});

// Retrieve one student
app.get('/students/:id', async (req, res) => {
  const student = await Student.findById(req.params.id);
  res.json(student);
});

// Update student
app.put('/students/:id', async (req, res) => {
  const updated = await Student.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(updated);
});

// Delete student
app.delete('/students/:id', async (req, res) => {
  await Student.findByIdAndDelete(req.params.id);
  res.status(204).send();
});
```

Each route corresponds to one CRUD operation, forming the backbone of the REST API.

Middleware and Error Handling. Middleware functions in Express allow developers to process requests before they reach route handlers. Common uses include authentication, logging, and validation. For example:

```
function logger(req, res, next) {
  console.log(`${req.method} ${req.url}`);
  next();
}

app.use(logger);
```

For error handling, Express provides a dedicated middleware pattern with four arguments:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Internal Server Error');
});
```

Centralized error handling ensures that the API can gracefully handle unexpected issues and provide meaningful feedback to clients.

Testing REST APIs. Once the server is running, REST endpoints can be tested using tools such as Postman, Insomnia, or the `curl` command-line utility. Testing ensures that each route responds with the correct data, status codes, and error messages.

Example test using curl:

```
curl -X POST http://localhost:3000/students \
     -H "Content-Type: application/json" \
     -d '{"name":"Ravi","age":23,"course":"Node.js"}'
```

The response should confirm the creation of the student resource and display the inserted document.

7.5.3 Hands-on — Practical Exercise

Practical Exercise: Building RESTful APIs with Express and MongoDB

1. **Initialize the project.** Create a new directory, initialize npm, and install dependencies:

```
mkdir express-api
cd express-api
npm init -y
npm install express mongoose
```

2. **Define the data model.** Create a file named `models/student.js`:

```
const mongoose = require('mongoose');
const studentSchema = new mongoose.Schema({
  name: String,
  age: Number,
```

```
  course: String
});
module.exports = mongoose.model('Student', studentSchema);
```

3. **Set up the Express server.** Create a file named `server.js`:

```
const express = require('express');
const mongoose = require('mongoose');
const Student = require('./models/student');

const app = express();
app.use(express.json());

mongoose.connect('mongodb://localhost:27017/school')
  .then(() => console.log('Database connected'))
  .catch(err => console.error(err));
```

4. **Implement CRUD routes.** Add routes for creating, retrieving, updating, and deleting students:

```
app.post('/students', async (req, res) => {
  const student = await Student.create(req.body);
  res.json(student);
});

app.get('/students', async (req, res) => {
  const students = await Student.find();
  res.json(students);
});

app.put('/students/:id', async (req, res) => {
  const updated = await Student.findByIdAndUpdate(req.params.id, req.body, { new: tru
  res.json(updated);
});

app.delete('/students/:id', async (req, res) => {
  await Student.findByIdAndDelete(req.params.id);
  res.sendStatus(204);
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

5. **Test the API.** Use Postman or `curl` to test all endpoints and confirm proper database interaction. Example test:

```
curl -X GET http://localhost:3000/students
```

6. **Enhance with middleware.** Add a simple logger or error-handling middleware to improve reliability and observability.
7. **Reflection.** Summarize the role of Express in simplifying web development, and explain how REST APIs provide a structured way for clients and servers to communicate.

7.6. Introduction to React

7.6.1 Objectives

In this section, students will learn how to build the frontend layer of a full-stack web application using the React library. The objective is to understand the fundamental building blocks of React—components, JSX syntax, state management, and props—and to use these effectively to construct interactive and dynamic user interfaces. Students will also learn to communicate with backend APIs using the Fetch API and Axios, as well as to handle forms and basic client-side validation. By the end of this section, learners will be able to create a simple React-based interface that interacts seamlessly with a Node.js and MongoDB backend.

7.6.2 Detailed Discussion

Overview of React. React is a JavaScript library developed by Meta (formerly Facebook) for building fast, interactive user interfaces. Unlike traditional page-based applications that reload the entire page with each change, React uses a *component-based* architecture and a *virtual DOM* (Document Object Model) to efficiently update only the parts of the interface that change. This makes React ideal for creating single-page applications (SPAs), where components update dynamically as users interact with the app.

Components: The Building Blocks of React. React applications are composed of reusable, self-contained components. Each component represents a distinct piece of the user interface—for example, a navigation bar, a form, or a data list. Components can be defined as JavaScript functions or ES6 classes, but most modern React development favors functional components due to their simplicity and compatibility with React Hooks.

A simple example of a React component:

```
function Welcome() {
  return <h1>Hello, React!</h1>;
}
```

Components can be nested, combined, and reused, allowing for modular and maintainable UI development.

JSX: Writing HTML in JavaScript. React uses a syntax extension called JSX (JavaScript XML), which allows developers to write HTML-like code inside JavaScript files. JSX enhances readability and integrates HTML structure with JavaScript logic.

Example:

```
function Greeting(props) {
  return <p>Welcome, {props.name}!</p>;
}
```

Here, `props.name` allows the component to receive data dynamically from its parent component.

Props and State. **Props** (short for properties) are read-only data passed from parent to child components, used to configure or customize behavior. **State** refers to mutable data managed within a component itself. When state changes, React automatically re-renders the component to reflect the new data.

Example of managing state:

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

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

  return (
    <div>
      <p>You clicked {count} times.</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

In this example, the `useState` hook initializes a variable `count` with a default value of zero and provides a function `setCount()` to update it.

Event Handling and Component Lifecycle. React supports event handling similar to HTML, but with JSX conventions. For example, the `onClick` attribute uses camelCase:

```
<button onClick={handleClick}>Submit</button>
```

Unlike traditional DOM manipulation, event handlers in React are integrated with component logic and automatically re-render UI elements when the underlying data changes.

Class components also provide lifecycle methods such as `componentDidMount()` and `componentWillUnmount()` to run code at specific points during a component's existence. In functional components, these are replaced by the `useEffect()` hook.

Fetching Data from APIs. React applications often need to fetch data from backend services. This is typically done using either the built-in Fetch API or a library like Axios. Example using Fetch:

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

function StudentList() {
```

```
  const [students, setStudents] = useState([]);

  useEffect(() => {
    fetch('http://localhost:3000/students')
      .then(response => response.json())
      .then(data => setStudents(data));
  }, []);

  return (
    <ul>
      {students.map(s => <li key={s._id}>{s.name}</li>)}
    </ul>
  );
}
```

Example using Axios:

```
import axios from 'axios';
useEffect(() => {
  axios.get('http://localhost:3000/students')
       .then(res => setStudents(res.data));
}, []);
```

These methods allow the frontend to interact with RESTful APIs developed with Express and MongoDB, achieving full-stack integration.

Form Handling and Validation. Handling user input in React involves managing form state and responding to change events. Example:

```
function AddStudent() {
  const [name, setName] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Submitted:', name);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter student name"
      />
      <button type="submit">Add</button>
    </form>
  );
```

```
}
```

Form validation can be implemented manually or with helper libraries such as `Formik` or `React Hook Form`, which simplify error messages, validation rules, and submission logic.

Through these foundational concepts, students will understand how React enables fast, maintainable, and interactive web interfaces that integrate smoothly with backend APIs.

7.6.3 Hands-on — Practical Exercise

Practical Exercise: Building a React Frontend

1. **Create a React application.** Install Vite or Create React App (CRA):

   ```
   npm create vite@latest react-frontend --template react
   cd react-frontend
   npm install
   npm run dev
   ```

   Verify that the default React page loads in the browser.

2. **Connect to the Express API.** Create a new component named `StudentList.jsx`:

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

   function StudentList() {
     const [students, setStudents] = useState([]);

     useEffect(() => {
       fetch('http://localhost:3000/students')
         .then(res => res.json())
         .then(data => setStudents(data));
     }, []);

     return (
       <ul>
         {students.map(s => <li key={s._id}>{s.name}</li>)}
       </ul>
     );
   }

   export default StudentList;
   ```

   Include this component in `App.jsx` to display student data retrieved from your backend.

3. **Add form handling.** Create a simple form to add new students:

```
function AddStudent() {
  const [name, setName] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();
    await fetch('http://localhost:3000/students', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ name })
    });
    alert('Student added!');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter student name"
      />
      <button type="submit">Add Student</button>
    </form>
  );
}
```

4. **Combine components.** Display the form and the list together in `App.jsx`:

```
import AddStudent from './AddStudent';
import StudentList from './StudentList';

function App() {
  return (
    <div>
      <h2>Student Management System</h2>
      <AddStudent />
      <StudentList />
    </div>
  );
}

export default App;
```

5. **Reflection.** Run your application, test the integration with your Express backend, and observe

how React updates the user interface dynamically as new students are added. Reflect on how component-based design and one-way data flow simplify application development.

7.7. Webpack and Modularization

7.7.1 Objectives

In this section, students will learn how to modularize JavaScript projects and bundle front-end applications efficiently using Webpack. The objective is to understand how Webpack transforms, optimizes, and manages application resources for both development and production environments. Students will also learn how to configure Babel for modern JavaScript (ES6+) and JSX compatibility, how to implement Hot Module Replacement (HMR) for real-time updates, and how to debug and optimize Webpack builds. By the end of this section, learners will be able to set up a complete build pipeline that transforms modular source code into optimized browser-ready bundles.

7.7.2 Detailed Discussion

The Need for Modularization. As web applications grow in size and complexity, managing large codebases with multiple JavaScript files, stylesheets, and assets becomes challenging. Modularization addresses this problem by dividing the code into smaller, reusable modules. Each module handles a specific functionality—such as rendering UI components, managing data, or configuring routes—and can be developed and maintained independently.

Modern JavaScript supports modularization natively through the ES6 `import` and `export` syntax, allowing developers to organize code logically and prevent global variable conflicts. For example:

```
// math.js
export function add(a, b) { return a + b; }

// app.js
import { add } from './math.js';
console.log(add(2, 3));
```

However, browsers do not natively understand module imports without additional tooling. This is where Webpack comes in.

What is Webpack? Webpack is a powerful module bundler for JavaScript applications. It takes multiple source files—JavaScript, CSS, images, fonts, and even JSON—and bundles them into one or more optimized output files. Webpack analyzes the dependency graph of an application and determines how modules depend on each other, ensuring they are loaded in the correct order during runtime.

The basic building blocks of a Webpack configuration are:

- **Entry:** The starting point of the application (typically `src/index.js` or `src/main.jsx`).
- **Output:** The destination folder and filename for the bundled code (often in `dist/`).

- **Loaders:** Tools that transform non-JavaScript files (like JSX, CSS, or images) into modules Webpack can understand.
- **Plugins:** Extensions that perform tasks such as optimization, minification, or environment variable injection.

Basic Webpack Configuration. A simple Webpack configuration file (`webpack.config.js`) for a React project might look like this:

```
const path = require('path');

module.exports = {
  entry: './src/index.jsx',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }
      },
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  resolve: {
    extensions: ['.js', '.jsx']
  },
  devServer: {
    static: './dist',
    hot: true
  },
  mode: 'development'
};
```

Here, Webpack processes JavaScript and JSX files through Babel, handles CSS imports, and serves the application using the Webpack development server with hot reloading enabled.

Babel and JSX Transformation. Babel is a JavaScript compiler that converts modern ES6+ and JSX syntax into backward-compatible JavaScript that browsers can understand. When working with React, Babel allows the use of JSX syntax and ES module imports seamlessly.

To use Babel with Webpack, install the necessary dependencies:

```
npm install --save-dev @babel/core @babel/preset-env @babel/preset-react babel-loader
```

Then, create a configuration file named `.babelrc`:

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

This setup ensures that React's JSX and modern JavaScript features are automatically transpiled during the build process.

Frontend and Backend Modularization. Webpack is typically used for front-end builds, while Node.js handles back-end modularization through its own CommonJS or ES module system. Separating front-end and back-end modules ensures cleaner code organization, improved scalability, and faster build times. For example:

- The **frontend module** (React) handles the user interface, components, and routing.
- The **backend module** (Express + Node.js) handles APIs, business logic, and database interactions.

During deployment, the front-end bundle (from Webpack) can be served by the backend Express server or hosted independently.

Hot Module Replacement (HMR). Hot Module Replacement (HMR) allows developers to update modules in real-time without reloading the entire page. This improves development speed and preserves component state during changes. HMR is automatically supported by Webpack's Dev Server when the `hot` option is enabled.

Example usage in `webpack.config.js`:

```
devServer: {
  static: './dist',
  hot: true,
  port: 3000
}
```

When editing a React component, HMR replaces only the modified module instead of reloading the browser, significantly improving the development experience.

Debugging and Optimization. Webpack provides several modes for controlling optimization:

- `development` — enables source maps and readable code for easier debugging.
- `production` — minifies code, removes dead code, and optimizes bundles.

Enable source maps for debugging:

```
devtool: 'source-map'
```

Source maps link the bundled code to the original source files, allowing developers to view the original line numbers and variable names in the browser's developer tools.

By mastering these concepts, students will understand how Webpack streamlines the modern front-end workflow, automating repetitive tasks and ensuring optimized, modular builds for real-world deployment.

7.7.3 Hands-on — Practical Exercise

Practical Exercise: Setting Up Webpack for React

1. **Initialize the project.** Create a new React project manually (without CRA) and install required dependencies:

```
mkdir webpack-react-demo
cd webpack-react-demo
npm init -y
npm install react react-dom
npm install --save-dev webpack webpack-cli webpack-dev-server \
             babel-loader @babel/core @babel/preset-env \
             @babel/preset-react html-webpack-plugin \
             css-loader style-loader
```

2. **Create the project structure.** Organize your files as follows:

```
webpack-react-demo/
   src/
       index.jsx
       App.jsx
   dist/
       index.html
   .babelrc
   webpack.config.js
   package.json
```

3. **Configure Babel.** Add a Babel configuration file named `.babelrc`:

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

4. **Set up Webpack.** Create a `webpack.config.js` file with entry, output, and loader rules:

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.jsx',
  output: {
    path: path.resolve(__dirname, 'dist'),
```

```
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        use: 'babel-loader'
      },
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  plugins: [new HtmlWebpackPlugin({ template: './dist/index.html' })],
  resolve: { extensions: ['.js', '.jsx'] },
  devServer: { static: './dist', hot: true, port: 3000 },
  mode: 'development'
};
```

5. **Enable Hot Module Replacement.** Run the development server:

   ```
   npx webpack serve
   ```

   Make edits in `App.jsx` and observe that changes appear instantly without refreshing the browser.

6. **Build for production.** Switch the mode to `production` and generate the final bundle:

   ```
   npx webpack --mode production
   ```

   Inspect the generated `bundle.js` file in the `dist/` directory.

7. **Reflection.** Summarize how Webpack automates the build process, transforms JSX and ES6 code, and optimizes the application for deployment. Discuss how modularization improves maintainability and performance in large projects.

7.8. MERN Stack Integration

7.8.1 Objectives ›

In this module, students will learn how to integrate MongoDB, Express, React, and Node.js into a unified, full-stack web application—commonly known as the MERN stack. The objective is to connect the React-based frontend with the Express and Node.js backend and to manage seamless communication with the MongoDB database. Students will also explore how to handle environment variables securely, configure Cross-Origin Resource Sharing (CORS), and organize their projects with a clear and scalable directory structure. By the end of this section, learners will have developed a working full-stack application capable of performing complete CRUD operations through RESTful APIs.

7.8.2 Detailed Discussion

Understanding the MERN Architecture. The MERN stack is a powerful combination of four technologies that enable developers to build end-to-end JavaScript applications:

- **MongoDB** — a NoSQL database used for storing application data in flexible, document-oriented format.
- **Express.js** — a lightweight Node.js framework for building APIs and managing server-side routing.
- **React.js** — a front-end library for creating interactive user interfaces and handling the client-side rendering of data.
- **Node.js** — the runtime environment that executes JavaScript on the server and hosts the backend logic.

The key advantage of the MERN stack is that it allows developers to use a single programming language—JavaScript—across the entire development pipeline, both client and server.

Connecting React to Express. The first step in MERN integration is establishing communication between the React frontend and the Express backend. The frontend sends HTTP requests (typically using Fetch or Axios) to the backend's RESTful API endpoints. The backend then processes these requests, interacts with the MongoDB database, and returns the results as JSON responses.

For instance, the React frontend might fetch data from the backend with:

```
fetch('http://localhost:5000/api/todos')
  .then(res => res.json())
  .then(data => setTodos(data));
```

On the backend, Express defines corresponding routes:

```
app.get('/api/todos', async (req, res) => {
  const todos = await Todo.find();
  res.json(todos);
});
```

This client-server relationship forms the foundation of a full-stack web application.

Managing Environment Variables. To keep sensitive information secure—such as database connection URIs, API keys, or port numbers—developers use environment variables. In Node.js, this is commonly achieved with the `dotenv` package. For example:

```
npm install dotenv
```

Then, in your `server.js` file:

```
require('dotenv').config();
const PORT = process.env.PORT || 5000;
const MONGO_URI = process.env.MONGO_URI;
```

Store variables in a file named `.env`:

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/mernapp
```

Using environment variables ensures that configuration data can be easily changed without modifying source code, improving portability and security.

Enabling CORS. Since React and Express often run on different ports during development (e.g., React on 3000 and Express on 5000), browsers enforce same-origin policies that restrict cross-domain requests. To allow the frontend to access backend resources, you must enable CORS (Cross-Origin Resource Sharing) on the server.

Install the CORS middleware:

```
npm install cors
```

Use it in your Express app:

```
const cors = require('cors');
app.use(cors());
```

This configuration permits the React frontend to make HTTP requests to the backend API without encountering cross-origin errors.

Structuring the Full-Stack Project. A well-organized project structure separates backend and frontend code while maintaining clarity and modularity. A typical MERN project layout looks like this:

```
mern-app/
  backend/
     models/
     routes/
     controllers/
     server.js
     .env
  frontend/
     src/
     public/
     package.json
  package.json
```

The backend and frontend each have their own `package.json` files for dependency management, but they can be run together during development using the `concurrently` package.

Handling Asynchronous API Requests. Because API requests involve network operations, they must be handled asynchronously to avoid blocking execution. React's `useEffect()` hook or Axios promises can be used to fetch data asynchronously. Example:

```
import axios from 'axios';
useEffect(() => {
  const fetchData = async () => {
    const res = await axios.get('http://localhost:5000/api/posts');
    setPosts(res.data);
  };
  fetchData();
}, []);
```

On the backend, Express routes use asynchronous functions with `async/await`:

```
app.post('/api/posts', async (req, res) => {
  try {
    const post = new Post(req.body);
    await post.save();
    res.status(201).json(post);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
```

This pattern ensures that the entire request-response cycle—from frontend event to database operation—remains non-blocking and efficient.

End-to-End Data Flow. The MERN stack supports a complete data flow:

1. The user interacts with the React frontend (e.g., submits a form).
2. The frontend sends an HTTP request to an Express API endpoint.
3. Express validates the request and performs CRUD operations on MongoDB using Mongoose.
4. The backend sends a JSON response to the frontend.
5. React updates the user interface dynamically to reflect the latest data.

This seamless communication across all four layers is what makes MERN applications powerful and unified.

7.8.3 Hands-on — Practical Exercise

Practical Exercise: Building a Full MERN Application

1. **Initialize the MERN project.** Create a folder and set up both backend and frontend:

```
mkdir mern-todo
cd mern-todo
```

```
npx create-react-app frontend
mkdir backend
cd backend
npm init -y
npm install express mongoose cors dotenv
```

2. **Create the backend server.** Inside the backend directory, create `server.js`:

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
require('dotenv').config();

const app = express();
app.use(cors());
app.use(express.json());

mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error(err));

const todoSchema = new mongoose.Schema({
  task: String,
  completed: Boolean
});
const Todo = mongoose.model('Todo', todoSchema);

app.get('/api/todos', async (req, res) => {
  const todos = await Todo.find();
  res.json(todos);
});

app.post('/api/todos', async (req, res) => {
  const newTodo = await Todo.create(req.body);
  res.status(201).json(newTodo);
});

app.put('/api/todos/:id', async (req, res) => {
  const updated = await Todo.findByIdAndUpdate(req.params.id, req.body, { new: true }
  res.json(updated);
});

app.delete('/api/todos/:id', async (req, res) => {
  await Todo.findByIdAndDelete(req.params.id);
```

```
  res.status(204).send();
});

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

3. **Connect the React frontend.** In `frontend/src/App.jsx`, add:

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

function App() {
  const [todos, setTodos] = useState([]);
  const [task, setTask] = useState('');

  useEffect(() => {
    fetch('http://localhost:5000/api/todos')
      .then(res => res.json())
      .then(data => setTodos(data));
  }, []);

  const addTodo = async (e) => {
    e.preventDefault();
    const res = await fetch('http://localhost:5000/api/todos', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ task, completed: false })
    });
    const newTodo = await res.json();
    setTodos([...todos, newTodo]);
    setTask('');
  };

  return (
    <div>
      <h2>MERN Todo App</h2>
      <form onSubmit={addTodo}>
        <input
          value={task}
          onChange={(e) => setTask(e.target.value)}
          placeholder="Add a new task"
        />
        <button type="submit">Add</button>
      </form>
      <ul>
```

```
          {todos.map(t => (
            <li key={t._id}>
              {t.task} {t.completed ? '(done)' : ''}
            </li>
          ))}
        </ul>
      </div>
    );
}

export default App;
```

4. **Run both servers concurrently.** Install the `concurrently` package in the root directory:

   ```
   npm install concurrently
   ```

   Then, add a combined start script to the root `package.json`:

   ```
   "scripts": {
     "start": "concurrently \"npm start --prefix backend\" \"npm start --prefix frontend
   }
   ```

5. **Reflection.** Run the application, test CRUD operations, and observe the end-to-end data flow between React, Express, and MongoDB. Reflect on how full-stack integration creates a seamless development experience using a single programming language—JavaScript—across all layers.

7.9. Module 8: Debugging and Deployment

7.9.1 Objectives

In this final module, students will learn how to debug, optimize, and deploy complete MERN stack applications. The focus is on diagnosing common issues in both frontend and backend components, improving performance for production, and deploying applications to cloud platforms such as Render, Vercel, and MongoDB Atlas. By the end of this section, learners will have practical experience in end-to-end debugging workflows, production configuration, and deployment pipelines.

7.9.2 Detailed Discussion

Debugging in Node.js. Debugging is an essential part of development and maintenance. Node.js provides a built-in inspector that can be accessed through Chrome DevTools, enabling developers to set

breakpoints, step through code, and analyze variable states.

To start debugging a Node application, run:

```
node --inspect server.js
```

Then, open Chrome and navigate to `chrome://inspect`. Selecting "Open dedicated DevTools for Node" allows you to inspect the backend code interactively, view the call stack, and evaluate expressions.

Using IDEs such as Visual Studio Code makes this process even smoother—VS Code's built-in debugger can attach to running Node processes, allowing you to step through request handlers and asynchronous functions with visual feedback.

React Debugging Tools. For the frontend, React provides the **React Developer Tools** browser extension for Chrome and Firefox. It enables inspection of component hierarchies, state values, and props in real time. Developers can use it to trace how data flows through components, check rendering performance, and identify re-rendering issues caused by state updates.

React's error boundaries and developer warnings are also invaluable for identifying runtime errors. Adding an error boundary component can help capture exceptions without breaking the entire application:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError() {
    return { hasError: true };
  }
  render() {
    if (this.state.hasError) return <h2>Something went wrong.</h2>;
    return this.props.children;
  }
}
```

Logging and Monitoring. Proper logging helps developers trace errors, monitor system activity, and analyze performance. In Express applications, two commonly used logging tools are:

- **Winston:** A flexible logging library supporting multiple transports (console, file, HTTP).
- **Morgan:** A lightweight HTTP request logger middleware for Express.

To integrate them:

```
npm install winston morgan
```

Example setup:

```
const morgan = require('morgan');
const winston = require('winston');

const logger = winston.createLogger({
  transports: [new winston.transports.File({ filename: 'app.log' })]
});
```

```
app.use(morgan('dev'));
app.use((req, res, next) => {
  logger.info(`${req.method} ${req.url}`);
  next();
});
```

This approach combines real-time console output (via Morgan) with persistent file logging (via Winston), ensuring developers can track both development-time and post-deployment issues.

Optimizing React with Webpack. Before deploying a React application, optimization is crucial to reduce bundle size and improve performance. Webpack's production mode automatically minifies code, removes unused dependencies (tree-shaking), and optimizes module loading. To generate a production build:

```
npm run build
```

This command (in Vite or CRA-based projects) produces an optimized output in the `dist/` or `build/` directory, ready to be served by an Express backend or static hosting service. Additional optimizations include:

- Code-splitting using React's `lazy()` and `Suspense` for faster initial load.
- Using environment variables to toggle between development and production APIs.
- Caching and compression via Webpack plugins or server configuration.

Deployment of MERN Applications. Once the application is debugged and optimized, it is ready for deployment. Each MERN component can be hosted on different platforms, depending on project requirements:

- **Render or Railway:** Ideal for hosting Express/Node backends. They automatically deploy from GitHub repositories and support environment variables and build scripts.
- **Vercel or Netlify:** Commonly used for React frontends, providing continuous deployment and global CDN distribution.
- **MongoDB Atlas:** Cloud-based MongoDB service offering database hosting, clustering, and built-in monitoring.

Example workflow:

1. Push the project to GitHub.
2. Deploy the backend on Render using a `start` script like:

   ```
   "start": "node server.js"
   ```

3. Deploy the React frontend on Vercel by connecting the GitHub repo and configuring the build command (`npm run build`) and output directory (`build/`).
4. Update environment variables (e.g., API base URLs) to point to the deployed backend and MongoDB Atlas connection string.

This completes the full lifecycle: from local development, debugging, optimization, and testing, to cloud deployment.

7.9.3 Hands-on — Practical Exercise

Practical Exercise: Debugging and Deploying a MERN Application

1. **Debug backend issues.** Run your Express application with the Node debugger:

   ```
   node --inspect server.js
   ```

   Set breakpoints using Chrome DevTools or VS Code, inspect request parameters, and trace asynchronous functions for errors.

2. **Debug React frontend.** Install React Developer Tools in your browser. Inspect component state, props, and rendering hierarchy. Identify components that re-render unnecessarily and optimize them with memoization or dependency control in hooks.

3. **Add logging and monitoring.** Integrate Winston and Morgan in your backend for structured request and error logging. Create a log file and verify that logs are written correctly for each incoming request.

4. **Build for production.** Run:

   ```
   npm run build
   ```

   Verify that the generated bundle is minified and optimized. Serve the build folder through your Express backend to test the production environment locally.

5. **Deploy to cloud platforms.** Deploy your backend to Render or Railway, your frontend to Vercel, and connect your database to MongoDB Atlas. Ensure environment variables are correctly configured for production URLs and database URIs.

6. **Reflection.** Analyze the debugging and deployment process. Record any issues encountered, their solutions, and insights about optimizing a MERN application for real-world environments.

7.10. Capstone Project

Project Overview. The capstone project serves as the culmination of all modules covered in this course. Students will apply their knowledge of MongoDB, Express, React, Node.js, and Webpack to design, implement, and deploy a complete, production-ready MERN stack application. This project emphasizes both functionality and code quality, simulating a professional full-stack development environment.

Requirements. Each project must demonstrate the following core competencies:

- **Backend:** Build an Express REST API connected to MongoDB using Mongoose for CRUD operations and schema validation.
- **Frontend:** Create a React-based user interface with forms, lists, and dynamic updates using Axios or Fetch for API communication.
- **Build Tools:** Configure Webpack and Babel for module bundling and ES6/JSX transpilation.
- **Environment Configuration:** Manage environment variables for development and production using `dotenv`.
- **Deployment:** Deploy the backend to Render (or a similar Node hosting platform), the frontend to Vercel, and the database to MongoDB Atlas.
- **Debugging Workflow:** Integrate Winston and React DevTools for systematic debugging during both development and deployment.

Suggested Project Ideas. Students may choose or propose their own project, but common examples include:

1. A **Task Management System** allowing users to create, update, and organize tasks.
2. A **Blog Platform** where users can write, edit, and delete posts.
3. A **Student Record Portal** that manages course registration and academic records.
4. A **Product Catalog Application** featuring CRUD operations, search, and category filters.

Deliverables. At the end of the course, each student must submit:

- Source code hosted on GitHub (including both frontend and backend folders).
- A live deployed version accessible via Render/Vercel.
- A short technical report (2–3 pages) describing the architecture, tools used, debugging steps, and deployment process.

Evaluation Criteria. Projects will be evaluated based on:

- Functionality and correctness of CRUD operations.
- Code organization and adherence to modular structure.
- User interface design and responsiveness.
- Logging, debugging, and error handling practices.
- Deployment reliability and documentation clarity.

Outcome. Upon completing this capstone project, students will have gained practical, industry-relevant experience in building, testing, debugging, and deploying a full-stack JavaScript application using modern web technologies.

7.11. Capstone: Complete Sample Project — MERN Todo App

7.11.1 Overview ›

This worked example walks students through building a simple but complete MERN Todo application from scratch. The goal is to demonstrate the full lifecycle: project initialization, backend development with Express and Mongoose, frontend development with React (Vite), local development with concurrent servers, debugging, and deployment considerations.

The final app allows users to:

- Create todo items.
- Read/list todo items.
- Update (toggle completed) todo items.
- Delete todo items.

7.11.2 Project Structure (final)

```
mern-todo/
  backend/
    controllers/
      todoController.js
    models/
      Todo.js
    routes/
      todos.js
    .env.example
    package.json
    server.js
  frontend/
    index.html
    package.json
    src/
      main.jsx
      App.jsx
      api.js
      components/
        AddTodo.jsx
        TodoList.jsx
  package.json   (workspace root for concurrent start script)
```

7.11.3 Step 0 — Prerequisites

Before starting, ensure students have:

- Node.js (v16+ recommended) and npm installed.
- MongoDB locally (or a MongoDB Atlas account and cluster URI).
- Git (optional) for version control.

7.11.4 Step 1 — Create project folders and initialize root repo

```
mkdir mern-todo
cd mern-todo
git init                # optional
```

Create ‘backend‘ and ‘frontend‘ folders:

```
mkdir backend frontend
```

7.11.5 Step 2 — Backend: Express + Mongoose

2.1 Initialize backend

```
cd backend
npm init -y
npm install express mongoose cors dotenv
```

2.2 Create `.env.example`

```
# backend/.env.example
PORT=5000
MONGO_URI=mongodb://localhost:27017/mern_todo
```

(Students will copy to ‘.env‘ and fill in their URI or Atlas connection string.)

2.3 Create `server.js`

```
// backend/server.js
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
require('dotenv').config();

const todosRouter = require('./routes/todos');

const app = express();
app.use(cors());
app.use(express.json());

const PORT = process.env.PORT || 5000;
const MONGO_URI = process.env.MONGO_URI || 'mongodb://localhost:27017/mern_todo';

// Connect to MongoDB
mongoose.connect(MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB connected'))
```

```
.catch(err => {
  console.error('MongoDB connection error:', err.message);
  process.exit(1);
});

// Routes
app.use('/api/todos', todosRouter);

// Global error handler (simple)
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Internal Server Error' });
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

2.4 Create Mongoose model: `models/Todo.js`

```
// backend/models/Todo.js
const mongoose = require('mongoose');

const todoSchema = new mongoose.Schema({
  task: { type: String, required: true, trim: true },
  completed: { type: Boolean, default: false },
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Todo', todoSchema);
```

2.5 Create controller: `controllers/todoController.js`

```
// backend/controllers/todoController.js
const Todo = require('../models/Todo');

exports.getAll = async (req, res, next) => {
  try {
    const todos = await Todo.find().sort({ createdAt: -1 });
    res.json(todos);
  } catch (err) { next(err); }
};

exports.create = async (req, res, next) => {
  try {
    const { task } = req.body;
    const todo = await Todo.create({ task });
```

```
    res.status(201).json(todo);
  } catch (err) { next(err); }
};

exports.update = async (req, res, next) => {
  try {
    const { id } = req.params;
    const updated = await Todo.findByIdAndUpdate(id, req.body, { new: true });
    res.json(updated);
  } catch (err) { next(err); }
};

exports.remove = async (req, res, next) => {
  try {
    const { id } = req.params;
    await Todo.findByIdAndDelete(id);
    res.status(204).end();
  } catch (err) { next(err); }
};
```

2.6 Create routes: `routes/todos.js`

```
// backend/routes/todos.js
const express = require('express');
const router = express.Router();
const ctrl = require('../controllers/todoController');

router.get('/', ctrl.getAll);
router.post('/', ctrl.create);
router.put('/:id', ctrl.update);
router.delete('/:id', ctrl.remove);

module.exports = router;
```

2.7 Backend `package.json` (example)

After you ran `npm init -y`, update ‘package.json‘ scripts:

```
{
  "name": "mern-todo-backend",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "cors": "^x.x.x",
    "dotenv": "^x.x.x",
```

```
    "express": "^x.x.x",
    "mongoose": "^x.x.x"
  },
  "devDependencies": {
    "nodemon": "^x.x.x"
  }
}
```

Install nodemon for development:

```
npm install --save-dev nodemon
```

7.11.6 Step 3 — Frontend: React (Vite)

3.1 Create a Vite React app From project root:

```
cd ../frontend
npm create vite@latest . -- --template react
npm install
```

(If the command prompts interactively, choose a project name or use the command shown.)

3.2 Frontend file: `src/api.js` This small module centralizes API calls and the base URL.

```
// frontend/src/api.js
const BASE = import.meta.env.VITE_API_BASE || 'http://localhost:5000';

export const API = {
  getTodos: async () => {
    const res = await fetch(`${BASE}/api/todos`);
    return res.json();
  },
  addTodo: async (task) => {
    const res = await fetch(`${BASE}/api/todos`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ task })
    });
    return res.json();
  },
  toggleTodo: async (id, completed) => {
    const res = await fetch(`${BASE}/api/todos/${id}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ completed })
    });
```

```
    return res.json();
  },
  deleteTodo: async (id) => {
    return fetch(`${BASE}/api/todos/${id}`, { method: 'DELETE' });
  }
};
```

3.3 Frontend component: `src/components/AddTodo.jsx`

```
// frontend/src/components/AddTodo.jsx
import { useState } from 'react';

export default function AddTodo({ onAdd }) {
  const [task, setTask] = useState('');

  const submit = async (e) => {
    e.preventDefault();
    if (!task.trim()) return;
    await onAdd(task.trim());
    setTask('');
  };

  return (
    <form onSubmit={submit}>
      <input
        value={task}
        onChange={(e) => setTask(e.target.value)}
        placeholder="Enter a new task"
      />
      <button type="submit">Add</button>
    </form>
  );
}
```

3.4 Frontend component: `src/components/TodoList.jsx`

```
// frontend/src/components/TodoList.jsx
export default function TodoList({ todos, onToggle, onDelete }) {
  if (!todos || todos.length === 0) return <p>No items yet.</p>;

  return (
    <ul>
      {todos.map(t => (
        <li key={t._id}>
          <label>
            <input
              type="checkbox"
```

```
              checked={t.completed}
              onChange={() => onToggle(t._id, !t.completed)}
            />
            {t.task} {t.completed ? '(done)' : ''}
          </label>
          <button onClick={() => onDelete(t._id)}>Delete</button>
        </li>
      ))}
    </ul>
  );
}
```

3.5 Frontend root: `src/App.jsx`

```
// frontend/src/App.jsx
import { useEffect, useState } from 'react';
import { API } from './api';
import AddTodo from './components/AddTodo';
import TodoList from './components/TodoList';

export default function App() {
  const [todos, setTodos] = useState([]);

  const load = async () => {
    const data = await API.getTodos();
    setTodos(data);
  };

  useEffect(() => { load(); }, []);

  const handleAdd = async (task) => {
    const newTodo = await API.addTodo(task);
    setTodos(prev => [newTodo, ...prev]);
  };

  const handleToggle = async (id, completed) => {
    const updated = await API.toggleTodo(id, completed);
    setTodos(prev => prev.map(t => t._id === id ? updated : t));
  };

  const handleDelete = async (id) => {
    await API.deleteTodo(id);
    setTodos(prev => prev.filter(t => t._id !== id));
  };

  return (
```

```
    <div>
      <h2>MERN Todo</h2>
      <AddTodo onAdd={handleAdd} />
      <TodoList todos={todos} onToggle={handleToggle} onDelete={handleDelete} />
    </div>
  );
}
```

3.6 Frontend entry: `src/main.jsx`

```
// frontend/src/main.jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './index.css'; // optional

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

3.7 Frontend environment variable

Create ‘frontend/.env‘ (not committed) or ‘frontend/.env.local‘:

```
VITE_API_BASE=http://localhost:5000
```

3.8 Frontend `package.json` scripts (Vite default)

Vite creates a ‘package.json‘ with scripts:

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview"
}
```

7.11.7 Step 4 — Root workspace scripts for concurrent development

At repo root create a ‘package.json‘ for convenience:

```
{
  "name": "mern-todo",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "start": "concurrently \"npm run dev --prefix backend\" \"npm run dev --prefix frontend\"",
    "start:prod": "npm run start --prefix backend"
  },
  "devDependencies": {
```

```
    "concurrently": "^x.x.x"
  }
}
```

Install concurrently at root:

```
npm install --save-dev concurrently
```

Also ensure ‘backend/package.json‘ contains ‘"dev": "nodemon server.js"‘ for hot-reloading backend in development.

7.11.8 Step 5 — Run the application locally

1. Create `.env` in `backend` from `.env.example` and set `MONGO_URI` (or use an Atlas URI).
2. Start both servers:

```
npm run start   # runs concurrently from repo root
```

3. Visit the frontend (Vite) dev URL — typically `http://localhost:5173` (Vite console prints URL).
4. The app should list todos, add new ones, toggle completed, and delete.

7.11.9 Step 6 — Debugging tips

Backend

- Run ‘nodemon‘ for automatic restarts: `npm run dev` in ‘backend‘.
- Add ‘console.log()‘ in route handlers and controllers to trace inputs/outputs.
- Use Node inspector: `node --inspect-brk server.js` and open Chrome DevTools to set breakpoints.
- Check MongoDB connectivity—if using Atlas, ensure IP whitelist or 0.0.0.0/0 for classroom.

Frontend

- Use React Developer Tools to inspect component state and props.
- Open browser console for network requests and check fetch responses.
- If CORS errors occur, ensure backend uses ‘app.use(cors())‘ during development.

7.11.10 Step 7 — Production build & serve

Option A: Serve built frontend from Express (single deploy unit) 1. Build frontend:

```
cd frontend
npm run build
```

This produces ‘dist/‘ (Vite) or ‘build/‘ (CRA). Copy or move contents to ‘backend/public‘.

2. Serve static files in ‘backend/server.js‘ (add near top, after express setup):

```
const path = require('path');
app.use(express.static(path.join(__dirname, 'public')));
app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});
```

3. Start backend in production mode:

```
npm run start --prefix backend
```

Option B: Deploy frontend and backend separately

- Deploy the backend to a Node hosting platform such as Render, Railway, or Heroku. Configure the platform's environment settings to include `MONGO_URI` (the MongoDB Atlas connection string) and `PORT` (the listening port).
- Deploy the frontend to a static-hosting or frontend platform such as Vercel or Netlify. Configure the project's build command (for Vite: `npm run build`) and set the environment variable `VITE_API_BASE` to the deployed backend URL (for example, `https://api.example.com`).

7.11.11 Step 8 — Deployment checklist

- Add environment variables securely (do not commit '.env' files).
- Enable logging (Winston) and error monitoring (Sentry or similar) for production.
- Use HTTPS endpoints and secure cookies if authentication is added.
- Ensure MongoDB Atlas has proper network access rules and backups configured.

7.11.12 Complete Source Listing (compact reference)

For instructor convenience, here is a consolidated listing of the critical files (already provided above). Use this as the canonical reference during labs.

7.11.13 Extensions and Improvements (student challenges)

- Add user authentication (JWT) and per-user todo lists.
- Add pagination and search/filter on the backend (aggregation).
- Add optimistic UI updates on the frontend for perceived performance.
- Integrate Webpack or code-splitting to optimize initial bundle.
- Add tests (Jest for backend, React Testing Library for frontend).

7.11.14 Teaching Notes and Expected Student Deliverables

- Each student (or group) must submit the complete source code to GitHub with a well-documented `README.md` file explaining how to install dependencies and run the project locally.
- Provide a live deployment link to both the frontend and backend (or a single combined application) hosted on platforms such as Render, Vercel, or Netlify.
- Submit a 2–3 page technical report describing:

- The overall system architecture and component interactions.
- Debugging experiences and key issues encountered during development.
- The deployment process and configuration of environment variables.
- Screenshots of developer tools, logs, and the final deployed application.

7.11.15 Closing remarks

This sample MERN Todo App is intentionally small so students can focus on understanding the full-stack flow end-to-end. It provides a concrete foundation for building larger systems and for extending the application with authentication, file uploads, and real-time features (WebSockets). Encourage students to iterate, test, and document their changes as part of good engineering practice.