Program Name: BCA
Course Name & Code: Full-Stack Development-II & BCA57116
Class: BCA2024
Academic Session: 2025-26

Study Material

(Introduction to React JS)

Table of Contents

		1.11 What is Document Object Model (DOM)? 1.11.1 Why is the DOM Important? 1.11.2 Working with DOM Elements 1.11.3 DOM Manipulation 1.11.4 DOM Events 1.11.5 Asynchronous DOM Operations 1.11.6 Security and the DOM
		1.12 The Virtual DOM 1.12.1 The Power of the Virtual DOM 1.12.2 Optimizing VDOM Usage 1.12.3 Common Misconceptions About the Virtual DOM 1.12.4 Challenges and Limitations

1.1 Getting Started with React

In recent years, Single Page Applications (SPAs) have become very popular in web development.

Initially, websites were simple: every time you clicked something or wanted to see a change, the entire page had to reload. This was slow because it took a long time for the request to go to the server and for the new page to come back.

Then came AJAX (Asynchronous JavaScript and XML). AJAX allowed parts of a page to update without reloading the whole thing. This was a big improvement.

SPAs are the next step in this evolution. With an SPA, most of the website's content is sent to your browser just once when you first load the page. After that, only small bits of information are exchanged with the server when needed. This is often done using simple "APIs" on the server side.

The rise of SPAs has been supported by many JavaScript tools and frameworks like Ext JS, KnockoutJS, BackboneJS, AngularJS, EmberJS, and more recently, React and Polymer.

Now, let's explore how React fits into this picture and learn more about it.

1.2 What is React?

ReactJS focuses on the **visual part of a website**, what users actually see. Think of it as the "V" (for View) in how many web applications are structured. It's flexible and doesn't tell you exactly how to use it.

React creates a kind of **blueprint of your website's look**. It breaks down the user interface into small, independent pieces called **Components**. Each component handles both how it looks and the behind-the-scenes logic that makes it work. These components can also hold the data they need to show the current state of your app.

To keep things simple and efficient, React aims to **re-render the entire application conceptually**. However, it's very smart about it.

React is built on the idea that directly changing the web page (called **DOM manipulation**) is a slow process and should be done as little as possible. It also realizes that manually optimizing these changes can lead to complicated, repetitive code that's prone to errors.

To solve this, React gives developers a **virtual DOM**. This is like a lightweight copy of the actual web page in memory. When something changes, React first updates this virtual copy. Then, it cleverly compares the new virtual DOM with the old one and figures out only the absolute minimum changes needed on the *actual* web page.

React is also **declarative**. This means you describe what you want the UI to look like, and React figures out how to make it happen. When your data changes, React "refreshes" its understanding of the UI and only updates the parts that are different.

This straightforward approach to data flow and simple display logic makes developing with ReactJS easy to understand and work with.

1.3 Who uses React?

You've probably used React without even knowing it! Many popular services like **Facebook, Instagram, Netflix, Alibaba, Yahoo, E-Bay, Khan Academy, AirBnB, Sony, and Atlassian** use React for their websites. In a short amount of time, major internet companies have adopted React for their main products.

1.4 Prerequisites of React Js

Learning ReactJS effectively builds upon a foundation of core web development skills. While you can certainly jump into React and learn along the way, having a solid grasp of these prerequisites will make your learning journey much smoother and prevent frustration.
Here's a breakdown of the essential prerequisites for learning ReactJS:

- **HTML (HyperText Markup Language):**

Why it's important: React is all about building user interfaces, and those interfaces are ultimately rendered as HTML in the browser. You need to understand how HTML elements are structured, what semantic tags are, and how they interact to create a webpage.
What you should know:

- Basic HTML tags (div, p, h1, ul, li, button, input, etc.)
- Attributes (e.g., src, href, class, id)
- Semantic HTML (using tags like <header>, <footer>, <nav>, <article>, <section> appropriately)
- Form elements and their basic functionality.

- **CSS (Cascading Style Sheets):**

Why it's important: While React focuses on the logic and structure of your UI, CSS is what makes it look good. You'll be styling your React components, so a good understanding of CSS is crucial.
What you should know:

- CSS selectors (class, ID, tag, pseudo-classes)
- Box model (margin, padding, border, content)
- Display properties (block, inline, inline-block, flexbox, grid)
- Basic styling properties (colors, fonts, sizes, backgrounds)
- Responsive design concepts (media queries for different screen sizes).

- **JavaScript Fundamentals (the most crucial prerequisite):**

Why it's important: React is a JavaScript library. You'll be writing almost all your application logic in JavaScript. A strong understanding of core JavaScript concepts is non-negotiable.
What you should know:

- **Variables:** var, let, const and their differences.
- **Data Types:** Numbers, strings, booleans, arrays, objects.
- **Operators:** Arithmetic, comparison, logical, assignment.
- **Control Flow:** if/else statements, switch statements.
- **Loops:** for, while, forEach, map, filter, reduce. (Especially map for rendering lists in React!).
- **Functions:** Declaring functions, function expressions, parameters, return values, arrow functions (very common in React).
- **Objects:** Creating, accessing properties, methods, this keyword.
- **Arrays:** Common array methods (push, pop, splice, slice, map, filter, reduce).
- **Event Handling:** How events work in the browser (e.g., click, submit, change).

- **ES6+ (ECMAScript 2015 and beyond):**

Why it's important: Modern React development heavily utilizes features introduced in ES6 (also known as ECMAScript 2015) and subsequent versions. These features make your code cleaner, more concise, and more powerful.
What you should know:

- **Arrow Functions (=>):** Widely used for conciseness and this binding.
- **let and const:** Block-scoped variable declarations.

- **Destructuring Assignment:** Easily extracting values from arrays or properties from objects.
- **Template Literals (backticks `):** Easier string interpolation.
- **Classes:** While React traditionally used class components, functional components with Hooks are now dominant. Still, understanding basic class syntax can be helpful if you encounter older codebases or specific patterns.
- **Spread and Rest Operators (...):** For copying arrays/objects and handling function arguments.
- **Modules (import/export):** How to organize your code into separate files and share functionality.
- **Promises/Async/Await (basic understanding):** For handling asynchronous operations, especially when fetching data from APIs.

- **JSX (JavaScript XML):**

Why it's important: JSX is a syntax extension for JavaScript that allows you to write HTML-like structures directly within your JavaScript code. While not strictly JavaScript, it's how most React components are written.

What you should know:

- How JSX looks and how it combines HTML and JavaScript.
- Rules for writing JSX (e.g., single root element, className instead of class, self-closing tags).
- Embedding JavaScript expressions within JSX using curly braces {}.

- **Node.js and npm/Yarn (Package Manager):**

Why it's important: You'll use Node.js to run your React development server and npm (Node Package Manager, which comes with Node.js) or Yarn to manage project dependencies (installing React, other libraries, build tools, etc.).

What you should know:

- How to install Node.js.
- Basic npm or yarn commands (e.g., npm install, npm run dev, npm build).

- **Command Line Interface (CLI):**

Why it's important: You'll be interacting with your project through the terminal for tasks like creating new projects, installing packages, and running the development server.

What you should know:

- Basic commands: cd (change directory), ls/dir (list contents), mkdir (make directory), npm create vite@latest, etc.

1.5 Why use React Js

ReactJS is a **JavaScript library** (not a full framework like Angular) for building user interfaces (UIs). It's developed and maintained by Facebook (Meta) and a community of individual developers and companies.

Here are the primary reasons why developers and companies choose ReactJS:

1.5.1 Component-Based Architecture:

What it is: React encourages you to break down your UI into small, isolated, reusable pieces called "components." Think of a website as a collection of LEGO bricks: a header is a component, a navigation bar is a component, a button is a component, a product card is a component.

Why it's beneficial:

- **Reusability:** You write code once and can use the same component across different parts of your application, or even in different projects. This saves development time and ensures consistency.

- **Maintainability:** Each component is self-contained. If something breaks in a specific UI part, you know exactly which component to look at, making debugging easier.
- **Collaboration:** Different developers can work on different components simultaneously without stepping on each other's toes.
- **Readability:** The code becomes more organized and easier to understand, as each file typically corresponds to a specific UI piece.

1.5.2 Declarative Programming:

What it is: With React, you describe *what* you want your UI to look like based on your data, rather than *how* to achieve that state. You tell React, "When this data changes, make the UI look like this."

Why it's beneficial:

Simpler UI Updates: You don't directly manipulate the DOM (Document Object Model, the browser's representation of your page). Instead, React handles the complex process of figuring out the most efficient way to update the actual browser UI.

Predictability: Because you declare the desired state, your UI behaves more predictably. When your data changes, you know exactly how the UI will react. This significantly reduces bugs related to manual DOM manipulation.

1.5.3 Virtual DOM (Document Object Model):

What it is: This is a core innovation of React. Instead of directly updating the browser's real DOM (which is a slow operation), React creates a lightweight "virtual" copy of the DOM in memory.

Why it's beneficial:

- **Performance:** When data changes, React first updates its virtual DOM. Then, it compares the current virtual DOM with the previous one to find the absolute minimum number of changes needed. Only these minimal changes are then applied to the actual browser DOM. This highly optimized process makes React applications very fast and responsive, especially for complex UIs with frequent updates.
- **Efficiency:** Developers don't have to worry about optimizing DOM updates manually, which is typically complex, error-prone, and time-consuming.

1.5.4 One-Way Data Flow (Unidirectional Data Flow):

What it is: Data in React flows in a single direction, typically from parent components down to child components via "props."

Why it's beneficial:

- **Predictability and Debugging:** This makes it much easier to understand how data is being used and transformed in your application. If something goes wrong, you can trace the data flow more predictably, making debugging much simpler.
- **Stability:** Child components cannot directly modify the data passed to them as props, ensuring that a parent component's data remains consistent and is only changed by the parent itself.

1.5.5 Strong Community and Ecosystem:

What it is: React has a massive, active community of developers, extensive documentation, and a rich ecosystem of tools, libraries, and resources.

Why it's beneficial:

- **Support:** If you run into a problem, chances are someone else has faced it before, and a solution or helpful advice is readily available.
- **Libraries and Tools:** There are countless third-party libraries for routing (React Router), state management (Redux, Zustand, Recoil), styling, testing, and more, which accelerate development.
- **Learning Resources:** Abundant tutorials, courses, and articles make it easier for new developers to learn React.

1.5.6 React Native for Mobile Development:

What it is: React Native is a framework that allows you to use your React knowledge and JavaScript skills to build native mobile applications for iOS and Android.

Why it's beneficial:

Code Reusability: You can often reuse a significant portion of your React web application's logic (though not the UI components directly) for mobile development.

Developer Efficiency: Developers familiar with React can quickly transition to mobile app development without learning entirely new languages (like Swift/Objective-C for iOS or Java/Kotlin for Android).

Native Performance: React Native compiles to actual native UI components, offering a performance much closer to truly native apps compared to hybrid web-view solutions.

1.5.7 Backed by Facebook (Meta):

What it is: Facebook uses React extensively in its own products (Facebook.com, Instagram, WhatsApp, etc.).

Why it's beneficial:

- **Reliability and Longevity:** This backing ensures continuous development, maintenance, and investment in the library, making it a reliable choice for long-term projects.
- **Real-World Testing:** React is constantly being tested and refined in large-scale, production environments.

In essence, ReactJS empowers developers to build complex, highly interactive, and performant user interfaces with greater ease, organization, and scalability. Its focus on components, declarative syntax, and efficient DOM updates makes it a powerful and enjoyable tool for modern web development.

1.6 Essential Software and Tools for ReactJS Development

To build, run, and manage a ReactJS project, you'll need a few key pieces of software:

1.6.1 Node.js (and npm/Yarn)

Description: Node.js is a JavaScript runtime environment that allows you to execute JavaScript code outside of a web browser. It's crucial for React development because:

- It powers the **development server** that runs your React application locally.
- It includes **npm (Node Package Manager)**, which is the standard tool for managing project dependencies. You'll use npm (or a faster alternative like Yarn or pnpm) to install React itself, other libraries, build tools, and more.

Why you need it: React projects rely on a build process and package management. Node.js provides the environment for these tools to run.

How to get it: Download from the official Node.js website (nodejs.org). It typically comes bundled with npm.

1.6.2 Code Editor (e.g., VS Code, Sublime Text, Atom)

Description: This is where you'll write all your React code (JavaScript, JSX, CSS, HTML). A good code editor offers features like:

- **Syntax highlighting:** Makes your code readable.
- **Autocompletion/IntelliSense:** Suggests code as you type.
- **Linting:** Helps you find errors and enforce coding styles.
- **Integrated Terminal:** Allows you to run commands without leaving the editor.
- **Extensions:** For React-specific snippets, formatting, and debugging.

Why you need it: It's your primary workspace for writing and managing your project files.

Recommendation: VS Code (Visual Studio Code) is by far the most popular and recommended choice for React development due to its excellent JavaScript/TypeScript support, vast extension marketplace, and integrated features.

1.6.3 Web Browser (with Developer Tools)

Description: A modern web browser like Google Chrome, Mozilla Firefox, Microsoft Edge, or Safari. All these browsers come with built-in "Developer Tools" (often accessible by pressing F12 or right-clicking and selecting "Inspect").

Why you need it:

- **Running your app:** Your React application ultimately runs in the browser.
- **Debugging:** Developer Tools are essential for inspecting HTML, CSS, running JavaScript, checking network requests, and debugging your React components.
- **React DevTools (Browser Extension):** This is a specialized browser extension (available for Chrome and Firefox) that allows you to inspect React component hierarchies, examine their props and state, and even modify them on the fly for debugging purposes. It's an invaluable tool for any React developer.

How to get it: Most modern operating systems come with a browser. Install the React DevTools extension from your browser's extension store.

1.6.4 Terminal / Command Prompt

Description: This is the text-based interface you use to interact with your operating system and run commands.

Why you need it:

- **Creating projects:** You'll use commands like npm create vite@latest to start a new React project.
- **Installing packages:** npm install or yarn add
- **Running the development server:** npm run dev or npm start
- **Running tests:** npm test
- **Building for production:** npm run build

How to get it:

- **Windows:** Command Prompt, PowerShell, or Git Bash (comes with Git).
- **macOS/Linux:** Terminal (built-in).
- Most modern code editors like VS Code have an integrated terminal, which is very convenient.

1.6.5 Git (Version Control System)

Description: Git is a distributed version control system that helps you track changes in your code, collaborate with others, and revert to previous versions if needed.

Why you need it:

- **Tracking changes:** Keeps a history of all modifications to your codebase.
- **Collaboration:** Essential for working in teams, allowing multiple developers to work on the same project without overwriting each other's work.
- **Backup/Recovery:** Your code is safely stored and you can easily revert to any previous state.
- **How to get it:** Download from git-scm.com or install via your operating system's package manager.

1.7 How to Set Up (Install) a ReactJS Project

Before you begin, make sure you have the prerequisites in place, especially **Node.js** (which includes npm).

Step 1: Ensure Node.js and npm are Installed

Description: React development relies heavily on Node.js, a JavaScript runtime, and its package manager, npm (Node Package Manager). If you don't have them, you need to install Node.js first.
How to check: Open your terminal or command prompt and type:

```
node -v
npm -v
```

If you see version numbers (e.g., v20.10.0 for Node, 10.2.3 for npm), you're good to go. If not, download and install Node.js from nodejs.org.

Step 2: Open Your Terminal or Command Prompt
Description: All the installation commands will be executed in your command line interface.
How to do it:

- **Windows:** Search for "Command Prompt," "PowerShell," or "Git Bash" (if you have Git installed).
- **macOS / Linux:** Search for "Terminal."

Step 3: Navigate to Your Desired Project Location (Optional but Recommended)
Description: It's good practice to create your project within a specific folder, like a dev or projects folder. Use the cd (change directory) command to move to that location.
Example:

```
cd C:\Users\YourUser\Documents\dev  # On Windows
# OR
cd ~/projects                 # On macOS/Linux
```

Step 4: Create a New React Project Using Vite
Description: Vite is a fast build tool that sets up a lean and performant development environment for React. The command below tells npm to create a new project using Vite, specifically with the React template.
Command:

```
npm create vite@latest my-react-app -- --template react
```

- npm create vite@latest: This invokes the latest version of Vite to create a new project.
- my-react-app: This is the **name of your project folder**. You can replace it with any name you like (e.g., my-portfolio, todo-list-app).
- -- --template react: This tells Vite to set up the project specifically for React development. The extra -- is needed to pass arguments directly to Vite's underlying command.

What happens:

- Vite will ask a few quick questions (e.g., "Select a framework:" react, "Select a variant:" JavaScript). You can usually just hit Enter for the default options or select react and JavaScript.
- It will create a new directory (e.g., my-react-app) with a basic React project structure inside it.

Step 5: Navigate into Your New Project Directory
Description: After the project is created, you need to change your current directory in the terminal to your new project's folder.
Command:

```
cd my-react-app
```

(Replace my-react-app with your actual project name).

Step 6: Install Project Dependencies
Description: The project structure created by Vite includes a package.json file. This file lists all the necessary libraries and tools (like React, React DOM, Vite itself) that your project needs to function. This step downloads these dependencies.
Command:

```
npm install
```

What happens:

- npm reads package.json and downloads all the listed packages from the npm registry.
- These packages are saved in a new folder named node_modules inside your project directory. This can take a minute or two depending on your internet connection.

Step 7: Start the Development Server

Description: Once all dependencies are installed, you can start the development server. This server will compile your React code, host your application locally, and provide features like "Hot Module Replacement" (HMR), which automatically updates your browser when you save changes to your code.

Command:

```
npm run dev
```

What happens:

- Vite starts a local development server.
- It will typically display a message like Vite vX.Y.Z ready in Xms and provide a local URL (e.g., http://localhost:5173/ or http://127.0.0.1:5173/).
- Your default web browser might automatically open to this URL, displaying a "Vite + React" welcome page.

1.8 React Project Directory Structure and Code Snippets

Here's the common structure you'll see in a new React project created with Vite:

```
my-react-app/
├── node_modules/
├── public/
│   └── vite.svg
├── src/
│   ├── assets/
│   │   └── react.svg
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── .eslintrc.cjs
├── .gitignore
├── index.html
├── package.json
├── package-lock.json
├── README.md
├── vite.config.js
└── jsconfig.json
```

Now, let's describe the key files and folders with their corresponding code snippets.

1. index.html (The Entry HTML File)

Description: This is the single HTML file that the browser loads. It's the "root" where your entire React application will be mounted. You usually don't add much content here, as React handles all the UI. It acts as a blank canvas for your SPA.

Location: my-react-app/index.html

Key Code Snippet:

```
<!doctype html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <link rel="icon" type="image/svg+xml" href="/vite.svg" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Vite + React App</title>
 </head>
 <body>
  <div id="root"></div> <script type="module" src="/src/main.jsx"></script> </body>
</html>
```

Explanation: Notice the . This is the element where React will render your entire application. The <script type="module" src="/src/main.jsx"></script> line is crucial; it tells the browser to load your main JavaScript entry file.

2. src/main.jsx (The React Entry Point)

Description: This JavaScript file is the very first piece of your React application that gets executed. Its job is to:

- Import React and ReactDOM (the library that allows React to interact with the browser's DOM).
- Import your main App component.
- Find the HTML element with id="root" in index.html.
- Render your App component into that HTML element.

Location: my-react-app/src/main.jsx

Key Code Snippet:

```
import React from 'react';
import ReactDOM from 'react-dom/client'; // Specifically for web rendering
import App from './App.jsx'; // Import your root App component
import './index.css'; // Import global styles

// Create a React root to manage updates
ReactDOM.createRoot(document.getElementById('root')).render(
 <React.StrictMode>
  <App /> {/* Render your App component here */}
 </React.StrictMode>,
);
```

Explanation: ReactDOM.createRoot() creates a React root. .render() then tells React to display the <App /> component (and all its children) inside the DOM element specified (in this case, the one with id="root"). <React.StrictMode> is a development-only helper for identifying potential problems in your application.

3. src/App.jsx (The Root Component)

Description: This is your main application component. It's typically where you compose other smaller components to build the overall layout and functionality of your application.

Location: my-react-app/src/App.jsx

Key Code Snippet:

```
import React from 'react'; // Not strictly needed in newer React for JSX, but good practice
import './App.css'; // Import its specific styles
import MyButton from './components/MyButton'; // Example: Importing a child component
```

```
import MessageDisplay from './components/MessageDisplay';
import { useState } from 'react'; // To manage state

function App() {
 const [buttonClicks, setButtonClicks] = useState(0);

 const handleButtonClick = () => {
  setButtonClicks(prevClicks => prevClicks + 1);
 };

 return (
  <div className="App"> {/* Using className instead of class for CSS */}
   <h1>Welcome to My React App!</h1>
   <p>This is the main application component.</p>

   <MyButton onClick={handleButtonClick} buttonText="Click Me!" />
   <MyButton onClick={handleButtonClick} buttonText="Another Button" />

   <MessageDisplay clickCount={buttonClicks} />

   {/* You can add more components and logic here */}
  </div>
 );
}

export default App; // Export the component so main.jsx can import it
```

Explanation:

- It's a JavaScript function (function App()).
- It uses useState to manage its internal buttonClicks state.
- It returns **JSX**, which looks like HTML but is JavaScript.
- It imports and uses other custom components like <MyButton /> and <MessageDisplay />, passing data to them using props.
- export default App; makes it available for main.jsx to import.

4. src/components/ (A Common Folder for Reusable Components)

Description: While not automatically created by Vite, it's a very common convention to create a components folder inside src/ to organize your reusable UI components.

Location Example: my-react-app/src/components/

Example Code Snippet: src/components/MyButton.jsx

```
import React from 'react';
import './MyButton.css'; // Optional: Component-specific CSS

function MyButton(props) {
 // Destructure props for cleaner code
 const { onClick, buttonText } = props;

 return (
```

```
    <button className="my-button" onClick={onClick}>
      {buttonText}
    </button>
  );
}

export default MyButton;
```

Explanation: This component receives onClick and buttonText as props from its parent (App.jsx). It renders a <button> element with the provided text and attaches the onClick function.

Example Code Snippet: src/components/MessageDisplay.jsx

```
import React from 'react';

function MessageDisplay(props) {
  const { clickCount } = props;

  let message = '';
  if (clickCount === 0) {
    message = 'No clicks yet.';
  } else if (clickCount === 1) {
    message = 'You clicked once!';
  } else if (clickCount > 1 && clickCount <= 5) {
    message = `You clicked ${clickCount} times. Keep going!`;
  } else {
    message = `Wow, you've clicked ${clickCount} times! Impressive.`;
  }

  return (
    <p>
      <strong>Status:</strong> {message}
    </p>
  );
}

export default MessageDisplay;
```

Explanation: This component receives clickCount as a prop and uses conditional logic to display a different message based on its value.

5. src/App.css and src/index.css (Styling)

Description: These are standard CSS files. index.css typically contains global styles (e.g., body fonts, universal resets), while App.css (and other component-specific .css files) contain styles local to that component.

Location Examples:

- my-react-app/src/index.css
- my-react-app/src/App.css
- my-react-app/src/components/MyButton.css

Key Code Snippet (Example App.css):

```
.App {
  font-family: Arial, sans-serif;
  text-align: center;
  padding: 20px;
}

h1 {
  color: #333;
}
```

Explanation: Standard CSS rules that apply to elements with the App class (which is applied to the root div in App.jsx).

6. package.json (Project Manifest)

Description: This file is a central manifest for your project. It lists project metadata, scripts for common tasks, and, most importantly, all your project's dependencies.

Location: my-react-app/package.json

Key Code Snippet (Example package.json):

```
{
  "name": "my-react-app",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "devDependencies": {
    "@types/react": "^18.2.43",
    "@types/react-dom": "^18.2.17",
    "@vitejs/plugin-react": "^4.2.1",
    "eslint": "^8.55.0",
    "eslint-plugin-react": "^7.33.2",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.5",
    "vite": "^5.0.8"
  }
}
```

Explanation:

- "scripts": Define commands you can run like npm run dev (starts development server) or npm run build (creates production build).

- "dependencies": Libraries required for the app to run (React, React DOM).
- "devDependencies": Libraries needed for development (Vite, ESLint).

Other Important Files/Folders:

While less frequently modified directly, these are crucial for your project's operation:

- **my-react-app/ (Root Directory):** This is your main project folder, containing everything.
- **node_modules/:** This folder holds all third-party libraries and packages installed via npm. You generally **never interact with this folder directly**; it's managed by npm and is excluded from version control.
- **public/:** Contains static assets (like vite.svg or favicon.ico) that are served directly to the browser without being processed by Vite.
- **.eslintrc.cjs:** Configuration file for ESLint, which helps maintain code quality and style.
- **.gitignore:** Tells Git which files and folders (like node_modules/ or dist/) to ignore in your version control.
- **package-lock.json (or yarn.lock/pnpm-lock.yaml):** An auto-generated file that locks the exact versions of all installed dependencies for consistent builds across environments. You **commit this file but don't manually edit it**.
- **README.md:** A markdown file for basic project documentation.
- **vite.config.js:** The configuration file for Vite, allowing you to customize its build and development server behavior.
- **jsconfig.json (or tsconfig.json if using TypeScript):** Configuration for JavaScript (or TypeScript) project settings, aiding code editors with features like auto-imports and path aliases.

1.9 Hands on code session on react

Bappa and Rahul are ready to dive into some hands-on React coding!

Bappa: "Alright Rahul, let's get our hands dirty with some React. The best way to learn is by building. Before we write any code, we need to set up our development environment. The easiest way to get started with a new React project is by using a tool called **Vite**."

Rahul: "Vite? I thought create-react-app was the standard."

Bappa: "Good question! create-react-app was the go-to for a long time, but it's been deprecated in favor of faster and more modern build tools like Vite. Vite offers a much quicker development server and a snappier build process. Let's start by installing Node.js if you don't already have it, as it includes npm (Node Package Manager), which we'll use for managing our project's dependencies."

Hands-on Session: Setting up our First React Project with Vite

Bappa: "First, open your terminal or command prompt."

Step 1: Create a new Vite React project

Bappa: "Run this command to create a new React project. Replace my-first-react-app with whatever name you want for your project."

npm create vite@latest my-first-react-app -- --template react

Rahul: "What does --template react do?"

Bappa: "That tells Vite to set up a project specifically for React, giving us the basic files and configurations we need for a React application. It also automatically selects JavaScript for the variant. After you run this, you'll see a few prompts. Just hit Enter for the defaults."

Step 2: Navigate into your project directory

Bappa: "Now, change into the newly created project folder:"

cd my-first-react-app

Step 3: Install project dependencies

Bappa: "Next, we need to install all the necessary packages and libraries for our React app. This will read the package.json file and download everything listed under dependencies."

npm install

Rahul: "This is taking a bit, lots of files downloading."

Bappa: "That's normal! It's pulling down React itself, React DOM (for rendering React to the browser), and other essential tools. Once it's done, you'll see a node_modules folder in your project."

Step 4: Start the development server

Bappa: "Finally, let's fire up our development server. This will open your React app in your browser."

npm run dev

Rahul: "Whoa! It opened a new tab in my browser and I see 'Vite + React'!"

Bappa: "Exactly! That's your first React application running locally. You can see the file structure in your project folder now. The most important files for us will be in the src directory, specifically App.jsx (which is our main component) and main.jsx (where our React app is rendered into the HTML)."

Hands-on Session: Modifying our First Component

Bappa: "Let's open up src/App.jsx in your code editor. This is where we'll start writing our React code."

Rahul: "Okay, I see a lot of stuff in here already."

Bappa: "Don't worry about all of it for now. Let's simplify it. Delete everything inside the return statement of the App function and replace it with a simple <h1> tag."

src/App.jsx (before)

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
```

```
import './App.css'
function App() {
 const [count, setCount] = useState(0)
 return (
  <>
   <div>
    <a href="https://vitejs.dev" target="_blank">
     <img src={viteLogo} className="logo" alt="Vite logo" />
    </a>
    <a href="https://react.dev" target="_blank">
     <img src={reactLogo} className="logo react" alt="React logo" />
    </a>
   </div>
   <h1>Vite + React</h1>
   <div className="card">
    <button onClick={() => setCount((count) => count + 1)}>
     count is {count}
    </button>
    <p>
     Edit <code>src/App.jsx</code> and save to test HMR
    </p>
   </div>
   <p className="read-the-docs">
    Click on the Vite and React logos to learn more
   </p>
  </>
 )
}

export default App
```

src/App.jsx (after simplification)

```
import React from 'react'; // We'll add this back as a good practice, though not strictly needed in newer React for JSX.
import './App.css'; // Keep the CSS for now.

function App() {
 return (
  <div>
   <h1>Hello, React!</h1>
  </div>
 );
}

export default App;
```

Bappa: "Now, save the file. What do you see in your browser?"

Rahul: "It instantly changed to 'Hello, React!' That's cool, I didn't even have to refresh the page!"

Bappa: "That's **Hot Module Replacement (HMR)** in action, provided by Vite. It's a huge time-saver in development. Now, let's talk about **JSX**. The <h1>Hello, React!</h1> part you just wrote inside the JavaScript function – that's JSX. It looks like HTML but it's actually JavaScript syntax extension that allows us to write UI structures right within our JavaScript code. React then takes that JSX and translates it into actual DOM elements."

Hands-on Session: Creating a Simple Component

Bappa: "Let's create our own custom component. In the src folder, create a new file called Button.jsx."

src/Button.jsx

```
import React from 'react';
function Button() {
  return (
    <button>
      I'm a simple button!
    </button>
  );
}
export default Button;
```

Bappa: "Now, we need to use this Button component inside our App component. Go back to src/App.jsx."

src/App.jsx (updated)

```
import React from 'react';
import './App.css';
import Button from './Button'; // Import our new Button component
function App() {
  return (
    <div>
      <h1>Hello, React!</h1>
      <Button /> {/* Use our Button component */}
      <p>This is a paragraph.</p>
    </div>
  );
}
export default App;
```

Bappa: "Save both files. What do you see now?"

Rahul: "I see 'Hello, React!', then a button that says 'I'm a simple button!', and then 'This is a paragraph.' all on the page. So, Button /> is how you use a component?"

Bappa: "Spot on! That's how you render a component. It's similar to an HTML tag, but it starts with a capital letter to distinguish it as a React component. This shows the power of components: we've broken down our UI into smaller, reusable pieces. We can use this Button component multiple times if we want, and if we change the Button.jsx file, all instances of Button will update."

Hands-on Session: Passing Data with Props

Bappa: "Now, let's make our Button component more dynamic. We want to be able to change the text on the button. This is where **props** come in. Props are how you pass data from a parent component (like App) to a child component (like Button)."

Bappa: "First, let's modify src/Button.jsx to accept a prop called text."

src/Button.jsx (with props)

```
import React from 'react';
function Button(props) { // Components receive props as an argument
 return (
  <button>
   {props.text} {/* Use the 'text' prop here */}
  </button>
 );
}
export default Button;
```

Rahul: "So, props is just an object?"

Bappa: "Precisely! It's a JavaScript object that contains all the properties passed to the component. Now, let's pass a text prop from App.jsx."

src/App.jsx (passing props)

```
import React from 'react';
import './App.css';
import Button from './Button';
function App() {
 return (
  <div>
   <h1>Hello, React!</h1>
   <Button text="Click me!" /> {/* Pass the 'text' prop */}
   <Button text="Learn More" /> {/* Pass a different 'text' prop */}
   <p>This is a paragraph.</p>
  </div>
 );
}
export default App;
```

Bappa: "Save both files. What do you see now?"

Rahul: "Now I have two buttons! One says 'Click me!' and the other says 'Learn More'. This is really neat! So, I can customize components just by passing different props."

Bappa: "You got it! Props are read-only, meaning a component should never directly modify the props it receives. They are like arguments to a function. This helps keep data flow predictable and debugging easier.

For now, this is a great start. You've set up a React project, understood how components work, and how to pass data using props. In our next session, we'll explore **state**, which is how components manage their own changing data and make your applications interactive!"

1.10 Software Applications Built with ReactJS

1.10.1 Facebook.com (Meta Platforms Inc.)

Description: The pioneering social media platform. Facebook's core web interface, including the dynamic News Feed, real-time notifications, and interactive elements, is heavily built with ReactJS. This demonstrates React's capability to handle massive scale and complex, constantly updating UIs.

1.10.2 Instagram.com (Meta Platforms Inc.)

Description: A globally popular photo and video sharing social network. Instagram's responsive web interface, including its feed, stories, explore page, and profile sections, extensively utilizes ReactJS to provide a smooth and engaging user experience for visual content.

1.10.3 Netflix

Description: A dominant subscription streaming service offering a vast library of movies and TV shows. Netflix employs ReactJS for various parts of its user interface, particularly for fast navigation, interactive content Browse, and the smooth presentation of its extensive media catalog.

1.10.4 WhatsApp Web (Meta Platforms Inc.)

Description: The web-based companion to the widely used mobile messaging application. WhatsApp Web leverages ReactJS to create its real-time chat interface, ensuring quick message delivery and a responsive user experience directly within a browser.

1.10.5 Airbnb

Description: A leading online marketplace for unique accommodations and travel experiences. Airbnb's complex website, featuring intricate search functionalities, interactive maps, booking flows, and user dashboards, is built using ReactJS to deliver a seamless and intuitive user journey.

1.10.6 Discord

Description: A popular communication platform for communities, offering voice, video, and text chat. Discord's web and desktop applications (which use Electron, a framework that renders web technologies) extensively rely on ReactJS for their dynamic chat interfaces, user management, and real-time interactions in large communities.

1.10.7 Khan Academy

Description: A non-profit organization providing free, world-class educational resources. Khan Academy utilizes ReactJS to power its interactive lessons, exercises, personalized learning dashboards, and other engaging educational tools.

1.10.8 Uber Eats

Description: Uber's platform for online food ordering and delivery. ReactJS is used in various sections of the Uber Eats website to manage complex elements like restaurant listings, interactive menus, order tracking, and the overall ordering process.

1.10.9 Asana

Description: A web and mobile application designed to help teams organize, track, and manage their work. Asana extensively uses ReactJS to build its highly interactive task management interface, enabling real-time updates, drag-and-drop functionality, and collaborative workflows.

1.10.10 Dropbox (Parts of)

Description: A cloud storage service for file hosting and collaboration. Specific components and sections of Dropbox's web application are built with ReactJS, enhancing the user experience for file Browse, sharing, and other interactive elements.

1.10.11 Microsoft 365 (Components within applications like Outlook Web)

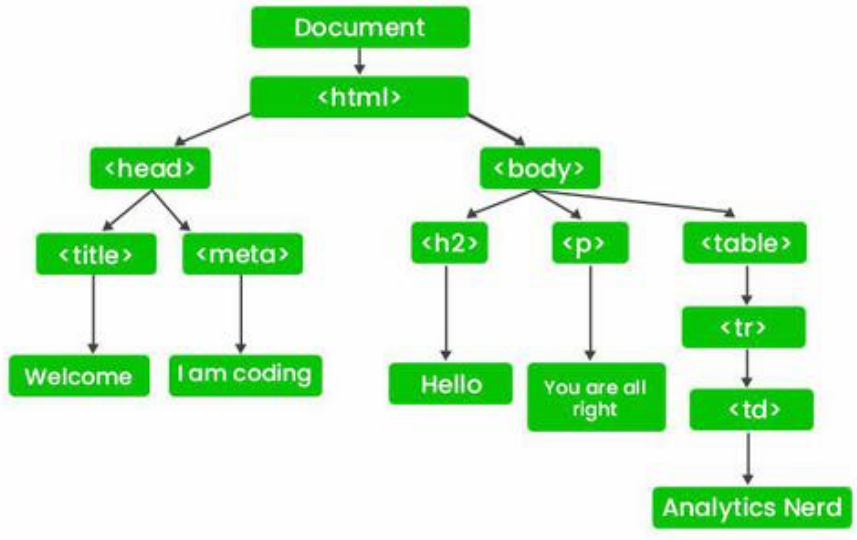
Description: Microsoft has integrated ReactJS into various web-based applications within its Microsoft 365 suite. For example, parts of the Outlook Web App utilize React to provide a modern, responsive, and efficient email and calendar experience.

1.10.12 Zendesk

Description: A prominent customer service software company that provides various tools for support and sales. Zendesk employs ReactJS in the development of its intuitive interfaces for customer support agents and customers interacting with their helpdesks and knowledge bases.

1.11 What is the Document Object Model (DOM)?

The **Document Object Model (DOM)** is a programming interface that acts as a **tree-like representation of a web page** in your browser's memory. When a browser loads an HTML document, it analyzes the content (tags, text, attributes) and transforms it into this structured object model. Each part of the HTML document – a heading, a paragraph, an image, a link – becomes a **node** or an **object** within this DOM tree, and these nodes are related to each other in a hierarchical structure.



1.11.1 Why is the DOM Important?

The DOM is fundamental because it provides **JavaScript** with the ability to **interact with and dynamically manipulate** the content, structure, and style of a web page. Without the DOM, a web page would be static. With it, developers can create **dynamic and interactive content** that responds to user actions. For instance, when a user clicks a button or submits a form, the page content can be updated without needing to reload the entire page, forming the basis of modern, responsive websites and applications.

1.11.2 Working with DOM Elements

JavaScript uses **DOM methods** to access and manipulate HTML elements. These methods allow you to select specific elements or groups of elements to then make changes.

getElementById(): Selects a single element based on its unique id attribute.

```
var element = document.getElementById("myElementId");
```

getElementsByClassName(): Selects all elements that belong to a specific class. It returns a live HTMLCollection.

```
var elements = document.getElementsByClassName("myClassName");
```

getElementsByTagName(): Selects all elements with a specific HTML tag name. It returns a live HTMLCollection.

```
var elements = document.getElementsByTagName("div");
```

querySelector() and querySelectorAll(): These are powerful methods that use CSS selectors to find elements.

querySelector(): Selects the **first** element that matches the given CSS selector.

```
var element = document.querySelector("#myElementId"); // Selects by ID
var firstParagraph = document.querySelector("p");    // Selects the first <p>
```

querySelectorAll(): Selects **all** elements that match the given CSS selector, returning a static NodeList.

```
var elements = document.querySelectorAll(".myClassName"); // Selects all elements with this class
var listItems = document.querySelectorAll("ul > li");    // Selects all <li> children of a <ul>
```

1.11.3 DOM Manipulation

Once elements are selected, JavaScript can **manipulate** them to change the page dynamically.

Changing Element Content: Use innerHTML (for HTML content) or textContent (for plain text) to modify an element's displayed content.

```
var element = document.getElementById("myElementId");
element.innerHTML = "<em>New bold content</em>"; // Changes HTML content
element.textContent = "New plain text";        // Changes plain text content
```

Creating New Elements: Add new HTML elements to the page dynamically.

```
var newElement = document.createElement("div"); // Creates a new <div>
newElement.textContent = "I'm a brand new item!";
document.body.appendChild(newElement);         // Appends it to the <body>
```

Removing Elements: Remove existing elements from the DOM. You typically need to access the parent element to remove a child.

```
var elementToRemove = document.getElementById("myElementId");
elementToRemove.parentNode.removeChild(elementToRemove);
```

Changing Element Styles and Properties: Directly modify an element's inline CSS styles or other HTML attributes.

```
var element = document.getElementById("myElementId");
element.style.color = "red";      // Changes text color
```

```
element.style.fontSize = "20px";  // Changes font size
element.setAttribute("aria-hidden", "true"); // Changes an attribute
```

1.11.4 DOM Events

Events are key to enabling user interaction. They represent user actions (like clicks, key presses, form submissions) or browser occurrences (like page loading). JavaScript allows you to "listen" for these events and react to them.

Event Listeners (addEventListener()): The standard way to attach functions that run when a specific event occurs on an element.

```
var button = document.getElementById("myButton");
button.addEventListener("click", function() {
  alert("Button clicked!");
});
```

Common Events:

- click: When an item is clicked.
- mouseover / mouseout: When the mouse pointer hovers over/leaves an element.
- keydown / keyup: When a keyboard key is pressed/released.
- submit: When a form is submitted.

Event Object: Event listener functions automatically receive an Event object, which contains useful information about the event (e.g., event type, the target element, mouse coordinates, key pressed).

1.11.5 Asynchronous DOM Operations

Modern web applications often retrieve data from servers (e.g., via APIs) and then update the DOM. These operations are typically **asynchronous** to prevent the web page from freezing ("blocking") while waiting for data. This ensures the page remains responsive.

Asynchronous Data Retrieval with Fetch API: The Fetch API is a promise-based method for making network requests.

```
fetch("https://api.example.com/data.json")
 .then(response => response.json()) // Parse the JSON response
 .then(data => {
  // Use the 'data' to update DOM elements
  console.log("Data fetched:", data);
  var myList = document.getElementById("myList");
  data.forEach(item => {
   var listItem = document.createElement("li");
   listItem.textContent = item.name;
   myList.appendChild(listItem);
  });
 })
 .catch(error => {
  console.error("Error fetching data:", error);
 });
```

1.11.6 Security and the DOM

When performing DOM manipulation, especially with user-provided content, **security is paramount**. Malicious users can exploit vulnerabilities to inject harmful code.

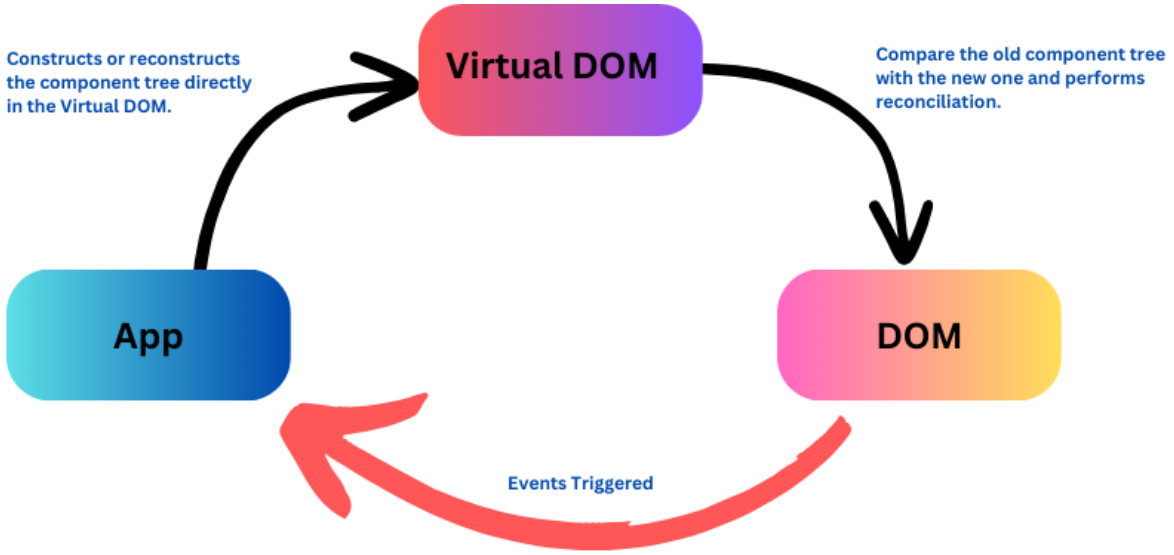
- **XSS (Cross-Site Scripting) Attacks:** These occur when malicious scripts are injected into web pages, often through user input that's then rendered in the DOM without proper sanitization.

- **Secure Data Handling:**
  - Always use safe methods like textContent or createElement() when inserting user input into the DOM, as these automatically escape potentially dangerous characters.
  - **Avoid using innerHTML with untrusted user data**, as it can execute injected scripts.
- **Content Security Policy (CSP):** A powerful browser security feature that helps prevent XSS and other attacks by controlling which resources the browser is allowed to load for a given page.
- **Secure Event Listeners:** Prefer addEventListener() over inline HTML event attributes (onclick="myFunction()") for better separation of concerns and often improved security.

1.12 The Virtual DOM

In ReactJS, there is the concept of a Virtual DOM. The main purpose of this is to enhance performance and efficiency in web development.

The Virtual DOM is a lightweight, in-memory representation of the real DOM. Instead of directly changing and manipulating the real DOM, which can be slow and less efficient due to extensive reflows and repaints, React creates a copy of the DOM where changes are initially made. This virtual replication allows ReactJS to minimize the number of direct manipulations applied to the actual browser DOM.



This virtual replication, the Virtual DOM, enables ReactJS to minimize direct and often costly manipulations of the actual browser DOM. When a state changes in an application, React first updates its internal Virtual DOM representation. It then efficiently compares this new Virtual DOM with the previous Virtual DOM tree using a "diffing" algorithm.

React's "diffing" process determines the precise differences between the two virtual trees. Subsequently, React updates *only* those specific parts of the actual, real DOM that have truly changed. This highly optimized "patching" approach, rather than re-rendering the entire DOM, significantly boosts the application's performance. It ensures a remarkably smooth and responsive user experience, even in highly dynamic and complex web applications with frequent state updates, as it avoids the performance bottlenecks associated with direct and extensive DOM manipulation.

The Virtual DOM (VDOM) is a lightweight, in-memory representation of the actual web page DOM, primarily used by frameworks like React to optimize UI updates. Instead of directly manipulating the slow real DOM, changes are first applied to the VDOM. React then uses a "reconciliation" process to compare the updated VDOM with its previous state, pinpointing only the necessary changes. These minimal changes are then efficiently applied to the real DOM, boosting performance in dynamic applications by avoiding unnecessary re-renders. This also enables a declarative programming style, where developers describe the desired UI state, and React handles the underlying DOM manipulations, further enhanced by its internal Fiber architecture for efficient, chunked rendering.

Here's a step-by-step breakdown of how the Virtual DOM works in React:

- **Initial Render:** When the application first loads, React constructs a complete Virtual DOM tree representing the entire UI based on your components.
- **State/Props Changes:** When a component's state or props update, React efficiently re-renders that component and its children into a *new* Virtual DOM tree. Crucially, these changes are not yet applied to the actual browser DOM.
- **Diffing Algorithm:** React then employs a highly optimized "diffing" algorithm to compare the newly generated Virtual DOM tree with the previous one. This process quickly identifies the precise differences (the "diffs") between the two versions.
- **Reconciliation:** Based on these identified differences, React determines the absolute minimum set of changes required to update the real DOM. It selectively updates only the changed elements, avoiding a full re-render of the entire UI. This intelligent, selective updating is the core of its performance advantage.
- **Real DOM Update:** Finally, React applies these calculated changes to the actual browser DOM. This might involve adding, removing, or updating specific elements as determined by the diffing process.

Example: Counter Functionality

Consider a simple React counter component:

```
import React, { useState } from 'react';
function App() {
 const [count, setCount] = useState(0);

 return (
  <div>
   <h1>Counter: {count}</h1>
   <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
 );
}

export default App;
```

Initially, when count is 0, the Virtual DOM representation would resemble:

```
{
 "type": "div",
 "props": {},
 "children": [
  {
   "type": "h1",
```

```
    "props": {},
    "children": [
     {
      "type": "TEXT_ELEMENT",
      "props": {
       "nodeValue": "Counter: 0"
      }
     }
    ]
   },
   {
    "type": "button",
    "props": {
     "onClick": "setCount(count + 1)" // Simplified for illustration
    },
    "children": [
     {
      "type": "TEXT_ELEMENT",
      "props": {
       "nodeValue": "Increment"
      }
     }
    ]
   }
  ]
}
```

When the "Increment" button is clicked and count becomes 1, React's diffing algorithm would identify that only the h1 element's nodeValue has changed. The resulting Virtual DOM update would effectively target just this part:

```
// Representing the detected change for the h1 element
{
 "type": "h1",
 "props": {},
 "children": [
  {
   "type": "TEXT_ELEMENT",
   "props": {
    "nodeValue": "Counter: 1"
   }
  }
 ]
}
```

React then efficiently updates *only* the text content within the h1 tag in the real DOM, leaving the rest of the page untouched.

1.12.1 The Power of the Virtual DOM

The VDOM offers several significant advantages:

- **Performance Optimization:** Directly manipulating the browser's Document Object Model (DOM) is inherently slow due to computationally intensive "reflow" and "repaint" processes. The VDOM mitigates this by batching updates and minimizing direct interactions with the real DOM. This leads to faster rendering and improved overall performance, especially in complex applications with frequent UI changes.
- **Declarative UI:** The VDOM enables a declarative programming style. Developers describe the desired UI state, and React efficiently updates the actual DOM to match this state. This abstraction simplifies development, making code more readable and easier to maintain.
- **Consistency and Predictability:** By managing all UI updates, the VDOM ensures that the application's UI consistently reflects its underlying state. This reduces the risk of unexpected changes or bugs that can arise from direct DOM manipulation, simplifying debugging and testing.
- **Efficiency through Reconciliation:** The VDOM's "diffing" algorithm precisely identifies changes between virtual representations. This "reconciliation" process ensures that only the necessary parts of the real DOM are updated, significantly reducing unnecessary operations and boosting efficiency.
- **Abstraction:** The VDOM abstracts away the intricacies of manual DOM manipulation, providing developers with a simpler interface. React handles the underlying optimization, leading to cleaner, more maintainable code.

1.12.2 Optimizing VDOM Usage

While the VDOM offers inherent benefits, achieving optimal performance requires developer understanding and proactive measures:

- **Understanding React's Lifecycle:** Effective VDOM utilization necessitates understanding React's component lifecycle, state management, and how updates propagate.
- **Performance Profiling:** Utilize tools to identify and address bottlenecks. Techniques like shouldComponentUpdate, PureComponent, and React.memo are crucial for optimizing re-renders.
- **Balancing Complexity and Maintainability:** Strive for a balance between leveraging the VDOM for efficiency and maintaining code simplicity. Best practices like component decomposition and thoughtful state management are key.

1.12.3 Common Misconceptions About the Virtual DOM

Several misconceptions often surround the Virtual DOM:

- **Not a Separate Browser API:** The VDOM is *not* a unique browser API. It's a lightweight, in-memory representation of the DOM managed entirely by libraries like React to optimize rendering.
- **Doesn't Always Guarantee Faster Performance:** While the VDOM generally optimizes rendering, it doesn't automatically make every React application faster. Performance gains depend heavily on

component structure and state management. Poorly designed components can still lead to performance issues.

- **Updates More Than Just Changed Elements:** The VDOM identifies *what* needs to change, but it doesn't *directly* update individual elements. Instead, it calculates the minimal set of changes and then applies them to the real DOM. This is more efficient than a full re-render but still involves DOM manipulation.
- **Doesn't Eliminate the Need for Optimizations:** The VDOM improves performance, but it's not a silver bullet. Developers still need to apply optimization techniques like memoization, avoiding unnecessary re-renders, and proper state management for peak performance.
- **Not Exclusive to React:** While React popularized the concept, the VDOM is also utilized by other frameworks and libraries like Vue.js and Inferno.
- **Compatible with Server-Side Rendering (SSR):** The VDOM works seamlessly with SSR. React renders components on the server using the VDOM, sending the pre-rendered HTML to the client, where React then takes over for client-side updates.

1.12.4 Challenges and Limitations

Despite its advantages, the Virtual DOM has certain challenges:

- **Memory Overhead:** Maintaining a VDOM alongside the actual DOM can increase memory consumption, especially in very large and complex applications.
- **Initial Render Time:** The initial rendering of a VDOM tree can sometimes be slower than rendering static HTML, as React must first build and reconcile the VDOM.
- **Reconciliation Complexity:** In deeply nested component structures or with extremely frequent updates, the diffing algorithm can become complex, potentially impacting performance.
- **Specific Performance Bottlenecks:** For certain scenarios, like handling massive lists or intricate animations, direct DOM manipulation or alternative optimized approaches might sometimes outperform the VDOM.
- **Event Handling and Asynchronicity:** React's asynchronous event handling and state updates can introduce complexity in managing updates across highly interactive components.