Front-End Development – II

HTML & CSS Revisited

HTML (HyperText Markup Language) and CSS (Cascading Style Sheets) are foundational technologies for building web pages.

Key Concepts:

1. **HTML Structure**:
   - Defines the structure of a webpage using elements such as <header>, <footer>, <section>, <article>, etc.
   - Example:

```
<html>
 <head>
  <title>My Website</title>
 </head>
 <body>
  <header>
   <h1>Welcome to My Website</h1>
  </header>
  <section>
   <p>This is a section of content.</p>
  </section>
 </body>
</html>
```

2. **CSS Styling**:
   - Provides styles for HTML elements (colors, layouts, typography).
   - Example:

```
body {
 font-family: Arial, sans-serif;
 background-color: #f4f4f4;
```

```
  color: #333;
}
h1 {
  text-align: center;
  color: #0066cc;
}
```

3. **Responsive Design with Media Queries**:
   - Adjusts styles based on screen size or device.
   - Example:

```
@media (max-width: 768px) {
  body {
    background-color: lightblue;
  }
}
```

Responsive Web Design Principles

Responsive web design ensures that websites work well on various devices and screen sizes.

Principles:

1. **Flexible Layouts**:
   - Use percentage-based widths instead of fixed pixels.
   - Example:

```
.container {
  width: 100%;
  max-width: 1200px;
  margin: 0 auto;
}
```

2. **Media Queries**:
   - Define styles for different screen sizes.
   - Example:

```
@media (max-width: 600px) {
  .menu {
    display: none;
  }
}
```

3. **Flexible Images and Media**:
    - Make images scalable.
    - Example:

```
img {
  max-width: 100%;
  height: auto;
}
```

4. **Mobile-First Design**:
    - Design for small screens first, then expand for larger screens.

Introduction to Bootstrap

Bootstrap is a popular front-end framework for building responsive and mobile-first websites.

Features of Bootstrap:

1. **Predefined Grid System**: Helps in layout design.
2. **Reusable Components**: Buttons, forms, modals, navigation bars, etc.
3. **Cross-Browser Compatibility**: Works consistently across different browsers.
4. **Customizable**: Easily customize themes and components.

Advantages of Bootstrap:

1. **Quick Development**: Saves time with ready-to-use components.
2. **Responsive Design**: Built-in responsive grid system.
3. **Consistent Design**: Ensures a uniform look and feel.
4. **Community Support**: Extensive documentation and active community.

Bootstrap Grid System

The grid system in Bootstrap allows you to create responsive layouts.

Key Concepts:

1. **Container**:
    - Centralizes the content.
    - Example:

```
<div class="container">
  <div class="row">
    <div class="col-md-3">Column 1</div>
    <div class="col-md-6">Column 2</div>
  </div>
</div>
```

2. **Rows and Columns**:
    - Divide the screen into 12 columns.
    - Example:

```
<div class="row">
  <div class="col-4">Column 1</div>
  <div class="col-4">Column 2</div>
  <div class="col-4">Column 3</div>
</div>
```

3. **Breakpoints**:
    - Adjust column widths for different screen sizes.
    - Breakpoints: col-sm-*, col-md-*, col-lg-*, etc.
    - Example:

```
<div class="row">
  <div class="col-md-8 col-sm-12">Main Content</div>
  <div class="col-md-4 col-sm-12">Sidebar</div>
</div>
```

Bootstrap Components

Bootstrap provides pre-built components for UI development.

1. **Buttons**:
   - Example:

```
<button class="btn btn-primary">Primary Button</button>
<button class="btn btn-danger">Danger Button</button>
```

2. **Forms**:
   - Example:

```
<form>
  <div class="form-group">
    <label for="email">Email:</label>
    <input type="email" class="form-control" id="email" placeholder="Enter email">
  </div>
  <button type="submit" class="btn btn-success">Submit</button>
</form>
```

3. **Navigation Bars**:
   - Example:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Brand</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
      <li class="nav-item"><a class="nav-link" href="#">Home</a></li>
      <li class="nav-item"><a class="nav-link" href="#">Features</a></li>
    </ul>
  </div>
```

```
</nav>
```

4. **Cards**:
    - Example:

```
<div class="card" style="width: 18rem;">
  <img src="image.jpg" class="card-img-top" alt="...">
  <div class="card-body">
    <h5 class="card-title">Card Title</h5>
    <p class="card-text">Some example text.</p>
    <a href="#" class="btn btn-primary">Go somewhere</a>
  </div>
</div>
```

1. Using Custom CSS

One of the simplest ways to customize Bootstrap is by overriding the default styles with your own CSS rules. This is useful when you want to make minor modifications without modifying Bootstrap's source files.

Example: Changing Button Style

By default, Bootstrap's `.btn-primary` has a blue background color (`#007bff`). You can override it using custom CSS:

```
.btn-primary {
  background-color: #ff6600; /* Change primary button to orange */
  border-color: #ff6600;
}
```

How to Apply:

- Create a new CSS file, e.g., `custom.css`
- Link it after Bootstrap's CSS file in your HTML:

```
<link rel="stylesheet" href="bootstrap.min.css">
<link rel="stylesheet" href="custom.css">
```

Pros:

- Quick and easy for small changes.
- No need to modify Bootstrap's source files.

Cons:

- If Bootstrap updates, your overrides may need to be re-checked.

2. SASS Customization

Bootstrap is built with **SASS**, which allows you to modify Bootstrap's variables before compiling CSS. Instead of overriding styles with CSS, you can directly modify Bootstrap's source variables.

Example: Changing Theme Colors

In `_variables.scss`, you can change Bootstrap's primary and secondary colors:

```
$primary: #007bff;  // Default primary color
$secondary: #6c757d; // Default secondary color

// Customizing:
$primary: #ff6600; // Change primary color to orange
$secondary: #333333; // Change secondary color to dark gray
```

How to Apply:

1. Install Bootstrap SASS (if using a project with SCSS support):

   ```
   npm install bootstrap
   ```

2. Import Bootstrap's SCSS file in your main SCSS file (`custom.scss`):

   ```
   @import "bootstrap/scss/bootstrap";
   ```

3. Compile the SCSS to CSS using a SASS compiler.

Pros:

- More control over Bootstrap's styles.
- Reduces CSS file size by modifying only necessary parts.
- Better maintainability than overriding styles in CSS.

Cons:

- Requires knowledge of SASS.
- Needs a build process for SCSS to CSS conversion.

3. Using Bootstrap Theme Builder

If you don't want to manually change styles using CSS or SCSS, you can use online tools like **Bootswatch, Bootstrap Theme Builder, or Bootstrap Magic** to generate customized Bootstrap themes.

Example: Using Bootswatch

1. Go to Bootswatch
2. Select a theme (e.g., "Darkly" or "Cosmo").
3. Download the modified Bootstrap CSS.
4. Replace the default Bootstrap CSS with the downloaded one in your project.

Pros:

- No coding required.
- Quick way to get a fresh Bootstrap look.

Cons:

- Limited customization options compared to SASS.
- May not always match project requirements exactly.

4. Selective Import (Tree Shaking)

Bootstrap provides modular SASS files, so you can **import only the required components** instead of including the entire Bootstrap CSS. This reduces the file size and improves performance.

Example: Importing Only Buttons

Instead of importing the full Bootstrap SCSS, you can selectively import specific parts:

```
@import "bootstrap/scss/functions";
@import "bootstrap/scss/variables";
@import "bootstrap/scss/buttons";
```

This approach **only includes button styles** and leaves out unnecessary styles like grids, forms, or alerts.

Pros:

- Improves performance by reducing unused CSS.
- Gives more control over what Bootstrap components are included.

Cons:

- Requires a SASS compiler.
- Might miss dependencies if not imported correctly.

1. Advance JavaScript (ES6 Concepts)

1.1 ES6 Classes

Classes in JavaScript were introduced in **ES6** to provide a more structured and cleaner way to create objects and handle inheritance.

Why Use Classes?

- Provides a **blueprint** for creating multiple objects with similar properties.
- Supports **inheritance** to create hierarchical relationships.
- Improves **code readability and maintainability**.

Example:

```
class Car {
    constructor(brand, model) {
        this.brand = brand;
        this.model = model;
    }

    display() {
        console.log(`Car: ${this.brand} ${this.model}`);
    }
}

const myCar = new Car("Tesla", "Model S");
myCar.display();
```

1.2 Arrow Functions

Arrow functions provide a **shorter syntax** for writing functions and do not have their own this binding.

Benefits of Arrow Functions :

✓ **Shorter syntax**

✓ **Lexical this binding** (inherits from parent scope)

✓ **No arguments object**

Example:

Const name = “abc”;

```
const greet = (name) => `Hello ${name}`
console.log(greet("Alice"));
```

1.3 Variables (let, const, var)

JavaScript has three ways to declare variables: var, let, and const.

Differences:

Feature	**var**	**let**	**const**
Scope	Function-scoped	Block-scoped	Block-scoped
Re-declaration	Allowed	Not allowed	Not allowed
Hoisting	Yes	No	No

Example:

```
let x = 10;
const y = 20;
x = 15;  // Allowed
// y = 30; ❌ Error: Cannot reassign a const variable
```

1.4 Array Methods

JavaScript provides **higher-order functions** to manipulate arrays efficiently.

Method	**Purpose**
map()	Transforms each array element.
filter()	Returns elements that meet a condition.
reduce()	Reduces array to a single value.

Example:

```
const numbers = [10, 20, 30, 40];
const filtered = numbers.filter(num => num > 20);
```

```
console.log(filtered); // [30, 40];
```

1.5 Destructuring

Destructuring allows extracting values from arrays or objects **easily**.

Example:

```
const person = { name: "Alice", age: 25 };
const { name, age } = person;
console.log(name, age);  // Alice 25
```

1.6 Spread Operator

The spread operator (...) is used to **expand arrays and objects**.

Example:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3 };
const merged = { ...obj1, ...obj2 };
console.log(merged); // {a: 1, b: 2, c: 3}
```

1.7 Modules

JavaScript modules help in organizing and **reusing** code.

Example (Exporting & Importing Modules):

```
// math.js
export const add = (a, b) => a + b;

// main.js
import { add } from './math.js';
console.log(add(2, 3));  // 5
```

1.8 Ternary Operator

A shortcut for if-else statements.

Example:

```
const age = 18;
const canVote = age >= 18 ? "Yes" : "No";
console.log(canVote);  //Yes
```

2. DOM (Document Object Model)

2.1 Introduction to DOM

The DOM (Document Object Model) represents an HTML document as a tree structure.

Why is DOM important?

- Modifies HTML content dynamically.
- Handles user interactions like clicks and keypresses.
- Manages animations in JavaScript.

2.2 Accessing DOM Elements

Example:

```
document.getElementById("myId");
document.getElementsByClassName("myClass");
document.getElementsByTagName("div");
document.querySelector(".myClass");
```

2.3 Event Handling

JavaScript handles user interactions using **events**.

Example:

```
document.getElementById("btn").addEventListener("click", function() {
   alert("Button Clicked!");
});
```

2.4 Dynamic DOM Manipulation

Creating New Elements

```
const newDiv = document.createElement("div");
newDiv.textContent = "Hello!";
document.body.appendChild(newDiv);
```

3. Async Programming in JavaScript

3.1 Introduction to Async Programming

Async programming allows JavaScript to execute **background tasks** without blocking the main thread.

Example (Blocking vs. Non-Blocking Code)

```
console.log("Start");
setTimeout(() => console.log("Delayed message"), 2000);
console.log("End");
```

Output:

```
Start
End
Delayed message
```

3.2 Timeout and Interval

Example (setTimeout):

```
setTimeout(() => console.log("Executed after 2 seconds"), 2000);
```

Example (setInterval):

```
setInterval(() => console.log("Repeats every 3 seconds"), 3000);
```

3.3 Promises

A **Promise** represents an asynchronous operation.

Example:

```
const fetchData = new Promise((resolve, reject) => {
   setTimeout(() => resolve("Data received"), 2000);
});
fetchData.then(data => console.log(data));
```

3.4 Async/Await

Async/Await makes asynchronous code look like **synchronous code**.

Example:

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

3.5 Callbacks

A **callback function** is passed as an argument to another function.

Example:

```
function fetchData(callback) {
    setTimeout(() => {
        callback("Data loaded");
    }, 2000);
}
fetchData(data => console.log(data));
```

3.6 Exception Handling in Async

Example:

```
async function fetchData() {
    try {
        let response = await fetch("https://invalid-url");
        let data = await response.json();
    } catch (error) {
        console.log("Error:", error);
    }
}
fetchData();
```

1. Introduction to Version Control

Version Control is a system for managing changes to files, typically source code, over time. It allows multiple people to work on a project simultaneously, tracks history, and lets you revert to previous versions if needed.

Benefits:

- **Collaboration:** Multiple developers can work on the same project without conflicts.
- **History Tracking:** Keeps track of every modification, enabling the restoration of previous versions.
- **Branching:** Facilitates parallel development by allowing the creation of separate branches.
- **Backup:** Acts as a backup by storing all versions of files.

Types of Version Control:

1. **Local Version Control Systems:**
   Store versions on a local machine using simple databases. Example: RCS (Revision Control System).
2. **Centralized Version Control Systems (CVCS):**
   Use a central server to store files and versions. Example: SVN (Apache Subversion).
   Drawbacks: If the server fails, all history can be lost.
3. **Distributed Version Control Systems (DVCS):**
   Every contributor has a complete copy of the repository, including history. Example: Git, Mercurial.
   Advantages: No single point of failure, better collaboration.

2. Version Control Using Git and Command Line

Git: A distributed version control system created by Linus Torvalds in 2005.

Key Git Concepts:

- **Repository:** A directory tracked by Git.
- **Commit:** A snapshot of changes.
- **Branch:** An independent line of development.
- **Merge:** Combining changes from different branches.

Common Commands:

- **Initialize a repository:**
  git init

- **Add files to staging area:**

  git add <filename> # Add a specific file

  git add . # Add all files
- **Commit changes:**

  git commit -m "Your commit message"
- **Undo last commit (soft reset):**

  git reset --soft HEAD~1

Example:

```
echo "Hello Git" > hello.txt
git init
git add hello.txt
git commit -m "Initial commit"
```

3. GitHub and Remote Repository

GitHub: A cloud-based Git repository hosting service.

Setting Up a Remote Repository:

1. Create a new repository on GitHub.
2. Link local repository to GitHub:

   git remote add origin https://github.com/username/repository.git
3. Push changes to GitHub:

   git push -u origin main

Cloning a Repository:

```
git clone https://github.com/username/repository.git
```

Listing Remote Repositories:

```
git remote -v
```

4. GitIgnore

.gitignore: A text file in your repository's root directory that tells Git which files to ignore.

Common .gitignore Patterns:

- **Ignore all .log files:**

  *.log
- **Ignore specific directories:**

  /node_modules

  /dist

- **Ignore environment files:**

  .env

Creating a .gitignore file:

```
echo "node_modules/" > .gitignore
git add .gitignore
git commit -m "Added .gitignore"
```

5. Git Cloning

Cloning: Creating a local copy of a remote repository including its history.

Command:

```
git clone <repository_url>
```

Example:

```
git clone https://github.com/username/repository.git
```

6. Branching and Merging

Branch: A separate version of the codebase to work on features independently.

Common Commands:

- **Create a new branch:**

  git branch feature-branch

- **Switch to a branch:**

  git checkout feature-branch

- **Merge branches:**

  git checkout main

  git merge feature-branch

Example Workflow:

```
git branch feature
git checkout feature
# Make some changes
git add .
git commit -m "Feature complete"
git checkout main
git merge feature
```

7. Forking and Pull Request

Fork: Creating a copy of someone else's repository under your GitHub account.

Pull Request: Request to merge changes from your fork into the original repository.

Workflow:

1. **Fork a repository** on GitHub.
2. **Clone your fork:**
   git clone https://github.com/yourusername/repository.git
3. **Make changes and commit.**
4. **Push changes to your fork:**
   git push origin main
5. **Create a pull request** on GitHub.

8. Introduction to APIs

API (Application Programming Interface): A set of rules that allow different software entities to communicate.

Key Components:

- **Endpoints:** URLs that provide access to resources.
- **Requests:** HTTP requests sent to endpoints.
- **Responses:** Data sent back by the API.

API Request Methods:

- **GET:** Retrieve data.
- **POST:** Create new data.
- **PUT:** Update existing data.
- **DELETE:** Remove data.

```
name: "iss",
id: 25544,
```

9. Structuring API Requests

Key Components of an API Request:

- **Endpoint:** URL of the API.
- **Method:** GET, POST, PUT, DELETE.
- **Headers:** Provide metadata (e.g., API keys).
- **Body:** Data sent with POST or PUT requests.

Example (Using Axios):

```
const axios = require('axios');
axios.get('https://api.example.com/data')
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

10. JSON (JavaScript Object Notation)

JSON: A format for structuring data, easily readable by humans and machines.

Example:

```
{
  "name": "Alice",
  "age": 28,
  "skills": ["JavaScript", "Python"]
}
```

11. Making Server-Side API Requests

Axios vs Fetch:

- **Axios:** Easier syntax, better error handling.
- **Fetch:** Native to browsers, returns a promise.

Axios Example:

```
axios.post('https://api.example.com/users', {
  name: 'John Doe',
  age: 30
})
.then(response => console.log(response.data))
.catch(error => console.error(error));
```

Fetch Example:

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ name: 'John Doe', age: 30 })
})
```

```
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

12. API Authentication

Common Authentication Methods:

- **API Key:** Simple but less secure.
- **OAuth:** Token-based, used for third-party access.
- **Bearer Token:** Sent in the header for secure access.

Example:

```
axios.get('https://api.example.com/data', {
 headers: {
  'Authorization': 'Bearer YOUR_API_KEY'
 }
});
```

13. RESTful APIs

Principles of REST:

1. **Uniform Interface:** Consistent endpoint design.
2. **Stateless:** No session data on the server.
3. **Cacheable:** Responses must be explicitly cacheable.
4. **Layered System:** Separation between client and server.

Example RESTful Endpoints:

```
GET /api/books        - Fetch all books
POST /api/books       - Add a new book
GET /api/books/1      - Fetch a book by ID
PUT /api/books/1      - Update a book
DELETE /api/books/1   - Delete a book
```

1. Introduction to React

React: A JavaScript library for building user interfaces, developed by Facebook. It enables developers to create fast, scalable, and simple front-end applications.

Key Features:

- **Component-Based Architecture:** Breaks UI into reusable components.
- **Easily handle the complex UI.**
- **Virtual DOM:** Efficiently updates and renders components by diffing with the actual DOM.
- **Declarative UI:** Describes what the UI should look like based on the application state.
- **Unidirectional Data Flow:** Ensures predictable state management.

Advantages of React:

1. **Reusability:** Components can be reused across the application.
2. **Performance:** Virtual DOM minimizes direct interaction with the real DOM.
3. **Strong Community Support:** Large ecosystem and libraries.
4. **SEO Friendly:** Server-side rendering capabilities.

2. Introduction to JSX and Babel

JSX (JavaScript XML): A syntax extension for JavaScript that looks similar to HTML but is converted to JavaScript by Babel.

Example of JSX:

```
const element = <h1>Hello, React!</h1>;
```

Babel: A JavaScript compiler that transforms JSX into plain JavaScript.

JSX to JavaScript Conversion:

```
const element = <h1>Hello, React!</h1>;
```

is converted to:

```
const element = React.createElement('h1', null, 'Hello, React!');
```

3. JSX Attributes and Styling React

Using Attributes in JSX:

- **Class becomes className:**

  ```
  <h1 className="header">Welcome</h1>
  ```

- **Self-closing tags:**

  ```
  <img src="logo.png" alt="Logo" />
  ```

Styling in React:

1. **Inline Styles:**
   ```
   <h1 style={{ color: 'blue', fontSize: '20px' }}>Hello, React!</h1>
   ```
2. **CSS Modules:**
   ```
   import styles from './App.module.css';
   <h1 className={styles.header}>Hello, React!</h1>
   ```
3. **Styled Components:** (Using a library)
   ```
   import styled from 'styled-components';
   const Button = styled.button`background-color: blue;`;
   ```

4. React Components

Components: The building blocks of React applications.

Types of Components:

1. **Functional Components:**
   - **Simpler and preferred for most cases.**
   - **Do not have their own state until React Hooks were introduced.**

   Example:
   ```
   function Greeting() {
    return <h1>Hello, React!</h1>;
   }
   ```
2. **Class Components:**
   - **Can have their own state.**
   - **Use lifecycle methods.**

   Example:
   ```
   class Greeting extends React.Component {
    render() {
      return <h1>Hello, React!</h1>;
    }
   }
   ```

5. State and Props

State: Managed within the component, used for dynamic data.

Using State in Functional Components:

```
import React, { useState } from 'react';
function Counter() {
```

```
  const [count, setCount] = useState(0);

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

Props: Data passed from one component to another.

Using Props:

```
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}

function App() {
  return <Welcome name="Alice" />;
}
```

6. React Hooks

Hooks: Special functions that let you use state and other React features in functional components.

Common Hooks:

1. **useState:** Manages state in functional components.

   Example:

   ```
   const [count, setCount] = useState(0);
   ```

2. **useEffect:** Runs side effects after rendering (similar to lifecycle methods).

   Example:

   ```
   import React, { useEffect } from 'react';
   function Timer() {
     useEffect(() => {
       const timer = setInterval(() => console.log('Tick'), 1000);
       return () => clearInterval(timer);
     }, []);
   ```

```
}
```

3. **useReducer:** Alternative to useState for complex state logic.

   Example:

```
import React, { useReducer } from 'react';
const initialState = { count: 0 };
function reducer(state, action) {
 switch (action.type) {
  case 'increment':
   return { count: state.count + 1 };
  case 'decrement':
   return { count: state.count - 1 };
  default:
   return state;
 }
}

function Counter() {
 const [state, dispatch] = useReducer(reducer, initialState);
 return (
  <div>
   <p>Count: {state.count}</p>
   <button onClick={() => dispatch({ type: 'increment' })}>+</button>
   <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
  </div>
 );
}
```

7. Lifecycle Methods in React

Lifecycle methods are only available in class components. They manage component initialization, rendering, updating, and unmounting.

Phases of Lifecycle:

1. **Mounting:** When the component is created and inserted into the DOM.
   - constructor()
   - componentDidMount()

2. **Updating:** When the component is re-rendered due to state or prop changes.
    - shouldComponentUpdate()
    - componentDidUpdate()
3. **Unmounting:** When the component is removed from the DOM.
    - componentWillUnmount()

Example:

```
class LifecycleDemo extends React.Component {
  constructor() {
    super();
    console.log('Constructor');
  }

  componentDidMount() {
    console.log('Component Mounted');
  }

  componentDidUpdate() {
    console.log('Component Updated');
  }

  componentWillUnmount() {
    console.log('Component Will Unmount');
  }

  render() {
    return <h1>React Lifecycle Methods</h1>;
  }
}
```

Summary of Key Differences: Functional vs. Class Components

Aspect	Functional Components	Class Components
Syntax	Simple functions	ES6 classes with render() method
State	useState hook	this.state

Aspect	Functional Components	Class Components
Lifecycle	useEffect hook	Lifecycle methods
Performance	Generally faster	Slightly slower due to this binding
Preferred Usage	For most scenarios	For legacy code

1. React Router: Client-Side Routing in React

React Router is a popular library for handling routing in React applications. It allows you to create multiple pages/views within a single-page application (SPA) without reloading the entire page.

Key Features of React Router:

1. **Client-Side Routing:** No full-page reloads.
2. **Nested Routes:** Define routes within routes.
3. **Dynamic Routing:** Routes that can accept parameters.
4. **Programmatic Navigation:** Navigate to different routes using code.

Installing React Router:

```
npm install react-router-dom
```

Basic Example:

```
import React from 'react';
import { BrowserRouter as Router, Route, Link, Routes } from 'react-router-dom';

function Home() {
 return <h2>Home Page</h2>;
}

function About() {
 return <h2>About Page</h2>;
}
```

```
function App() {
 return (
  <Router>
   <nav>
    <Link to="/">Home</Link> | <Link to="/about">About</Link>
   </nav>
   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
   </Routes>
  </Router>
 );
}

export default App;
```

Explanation:

- **<Router>:** Wraps the entire app to enable routing.
- **<Route>:** Defines a path and the component to render.
- **<Link>:** Allows navigation between routes without reloading.

Dynamic Routes Example:

```
import { BrowserRouter as Router, Route, Routes, useParams } from 'react-router-dom';

function UserProfile() {
 const { username } = useParams();
 return <h2>Welcome, {username}!</h2>;
}

function App() {
 return (
  <Router>
   <Routes>
    <Route path="/user/:username" element={<UserProfile />} />
```

```
      </Routes>
    </Router>
  );
}

export default App;
```

Explanation:

- **useParams()** retrieves route parameters, such as the username.

2. Managing Application State in React

State Management: The practice of managing and sharing state (data) across different parts of an application.

Ways to Manage State:

1. **Local State:** Managed using useState within individual components.
2. **Global State:** Managed using Context API or external libraries like Redux.

A. Using React Context API for Global State

Context API: A built-in feature to pass data through the component tree without prop drilling.

Step 1: Create a Context

```
import React, { createContext, useState } from 'react';

export const UserContext = createContext();

export function UserProvider({ children }) {
  const [user, setUser] = useState({ name: 'Alice', loggedIn: true });
  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
    </UserContext.Provider>
  );
}
```

Step 2: Use Context in Components

```
import React, { useContext } from 'react';
import { UserContext } from './UserProvider';

function Dashboard() {
 const { user } = useContext(UserContext);
 return <h2>Welcome, {user.name}!</h2>;
}

export default Dashboard;
```

B. Using Redux for State Management

Redux: A library for managing global state in a predictable way using actions, reducers, and a central store.

Key Concepts:

1. **Store:** Holds the global state.
2. **Actions:** Describe what happened.
3. **Reducers:** Specify how the state changes.

Example: Counter with Redux

1. Install Redux:

```
npm install redux react-redux
```

2. Create Reducer:

```
const initialState = { count: 0 };
function counterReducer(state = initialState, action) {
 switch (action.type) {
  case 'INCREMENT':
   return { count: state.count + 1 };
  case 'DECREMENT':
   return { count: state.count - 1 };
  default:
```

```
    return state;
  }
}

export default counterReducer;
```

3. Create Store:

```
import { createStore } from 'redux';
import counterReducer from './counterReducer';

const store = createStore(counterReducer);
export default store;
```

4. Dispatch Actions:

```
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'DECREMENT' });
```

3. Component Styling in React

A. Inline Styles:

```
function StyledComponent() {
  return <h1 style={{ color: 'blue', fontSize: '24px' }}>Hello, Styled World!</h1>;
}
```

B. CSS Modules:

- **App.module.css:**

```
.header {
  color: blue;
  font-size: 24px;
}
```

- **Using CSS Modules:**

```
import styles from './App.module.css';

function StyledComponent() {
  return <h1 className={styles.header}>Hello, CSS Modules!</h1>;
}
```

C. Styled-Components:

```
npm install styled-components
```

Example:

```
import styled from 'styled-components';

const Button = styled.button`
 background-color: blue;
 color: white;
 padding: 10px;
 border-radius: 5px;
`;

function App() {
 return <Button>Click Me</Button>;
}
```

4. Testing React Applications

Testing: Ensures that components work as expected.

Tools for Testing:

1. **Jest:** A testing framework for JavaScript.
2. **React Testing Library:** Helps in testing React components more effectively.

A. Setting Up Jest:

```
npx create-react-app my-app --template cra-template-pwa
npm install --save-dev jest
```

B. Writing Tests with Jest:

Component (Button.js):

```
export function Button({ label }) {
 return <button>{label}</button>;
}
```

Test (Button.test.js):

```
import { render, screen } from '@testing-library/react';
import { Button } from './Button';

test('renders the button with the correct label', () => {
  render(<Button label="Click Me" />);
  const buttonElement = screen.getByText(/Click Me/i);
  expect(buttonElement).toBeInTheDocument();
});
```

C. Running Tests:

```
npm test
```

Summary Table: Key Differences in State Management Approaches

Approach	Best For	Examples
Local State (useState)	Simple, component-specific	Form handling, toggles
Context API	Small to medium apps	User authentication, themes
Redux	Large, complex apps	E-commerce carts, multi-level data sharing