M.Sc. Computer Application

SYBCA – SEM 3

904-01 Application Development using UI

Table of Contents

Unit 1.

1.1 Building Blocks of Web Application Development:

The building blocks of web application development encompass several core components and technologies, each playing a vital role in creating a functional, user-friendly, and scalable web application. Here's a breakdown of these building blocks:

1. Frontend Development

- **HTML (HyperText Markup Language)**: The backbone of web content, HTML structures the content of the web pages.
- **CSS (Cascading Style Sheets)**: CSS handles the visual styling of the web application, including layouts, colors, fonts, and responsiveness.
- **JavaScript**: JavaScript adds interactivity and dynamic behavior to web pages, enabling features like form validation, animations, and user interface (UI) updates without page reloads.
- **Frontend Frameworks/Libraries**:
    - **React.js, Angular, Vue.js**: Popular libraries and frameworks for building complex, reactive UIs. They help manage the state and render components efficiently.
- **Responsive Design**: Ensuring the application works on various devices and screen sizes using techniques like media queries and responsive grid systems.

2. Backend Development

- **Server-side Programming Languages**:
    - **PHP, Python, Ruby, Java, JavaScript (Node.js), Kotlin**: These languages are used to write the backend logic, handle requests, and interact with databases.
- **Backend Frameworks**:
    - **Laravel (PHP), Django (Python), Spring (Java), Express (Node.js)**: Frameworks that provide structure to backend development, offering features like routing, authentication, and database interaction.
- **APIs (Application Programming Interfaces)**:
    - **RESTful APIs, GraphQL**: Methods for the frontend to communicate with the backend, enabling data exchange and service consumption.
- **Databases**:

  - **SQL Databases (MySQL, PostgreSQL), NoSQL Databases (MongoDB, Redis)**: Databases store, retrieve, and manage data for the application.

- **Server Management**:
  - **Nginx, Apache**: Web servers that handle HTTP requests, serve static content, and proxy requests to backend servers.

- **Security**:
  - **SSL/TLS, Authentication & Authorization, Data Encryption**: Ensuring data integrity, confidentiality, and secure user access.

3. DevOps & Deployment

- **Version Control**:
  - **Git, GitHub, GitLab**: Tools for source code management, enabling collaboration and version tracking.

- **CI/CD Pipelines**:
  - **Jenkins, CircleCI, GitHub Actions**: Automated tools for testing, building, and deploying applications.

- **Containerization & Virtualization**:
  - **Docker, Kubernetes**: Technologies for packaging applications and their dependencies into containers, facilitating consistent deployment across environments.

- **Cloud Services & Hosting**:
  - **AWS, Azure, DigitalOcean, Heroku**: Cloud platforms offering scalable infrastructure for hosting applications and databases.

- **Monitoring & Logging**:
  - **Prometheus, Grafana, ELK Stack**: Tools for tracking application performance, detecting errors, and maintaining logs for analysis.

4. Testing & Quality Assurance

- **Unit Testing**:
  - **Jest, Mocha, PHPUnit**: Tools for testing individual components or functions to ensure they work as expected.

- **Integration Testing**:
  - **Postman, SoapUI**: Testing the interaction between different parts of the application.

- **End-to-End Testing**:
    - **Cypress, Selenium**: Testing the entire application flow from the user's perspective.
- **Performance Testing**:
    - **JMeter, LoadRunner**: Tools for assessing application performance under load.

5. UX/UI Design

- **Wireframing & Prototyping**:
    - **Figma, Adobe XD, Sketch**: Tools for designing the user interface and user experience flow before development.
- **User Research & Testing**:
    - **A/B Testing, User Feedback**: Techniques for understanding user behavior and preferences to refine the application's design and functionality.

6. Project Management

- **Agile Methodologies**:
    - **Scrum, Kanban**: Frameworks for managing the development process, focusing on iterative progress and flexibility.
- **Collaboration Tools**:
    - **JIRA, Trello, Asana**: Tools for tracking tasks, managing projects, and collaborating among team members.

These components work together to build a robust, scalable, and user-friendly web application. Depending on the project's requirements, the stack might vary, but the core principles remain the same.

1.1.1 Single-page and Multi-page Applications, Different Client-side Technologies

Single-page Applications (SPAs)

A Single-page Application (SPA) is a web application that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from the server. This approach provides a more fluid user experience, similar to a desktop application.

Characteristics of SPAs:

- **Single HTML File**: SPAs load a single HTML file initially and then dynamically update the content as the user interacts with the app.
- **Client-side Routing**: SPAs use client-side routing to change the URL and render different views without a full-page reload.
- **Asynchronous Data Loading**: Data is loaded asynchronously using JavaScript (often via AJAX or Fetch API), reducing the need for full-page reloads.
- **Fast & Responsive**: By only updating parts of the page, SPAs offer a faster and more responsive user experience.
- **Examples**: Gmail, Google Maps, Facebook, Twitter.

Technologies Commonly Used in SPAs:

- **JavaScript Frameworks/Libraries**:
    - **React.js**: A JavaScript library for building user interfaces, particularly SPAs.
    - **Angular**: A full-featured framework that provides everything needed for building SPAs.
    - **Vue.js**: A progressive framework that can be incrementally integrated into a SPA.
- **Routing Libraries**:
    - **React Router (for React.js)**, **Vue Router (for Vue.js)**, **Angular Router (for Angular)**: These libraries manage the navigation between different views in SPAs.
- **State Management**:
    - **Redux, MobX, Vuex**: Libraries for managing the state of the application across different components.

Advantages of SPAs:

- **Faster Load Times**: Only the necessary data and components are loaded as the user interacts with the app.
- **Seamless User Experience**: No full-page reloads result in a smoother, more application-like experience.
- **Efficient Resource Usage**: Resources are loaded once and reused, which can reduce server load.

Disadvantages of SPAs:

- **SEO Challenges**: SPAs can be difficult to optimize for search engines since content is loaded dynamically.
- **Initial Load Time**: The first load can be slower due to the loading of the full JavaScript bundle.
- **Browser History Management**: Managing browser history and back button behavior can be more complex.

Multi-page Applications (MPAs)

A Multi-page Application (MPA) is a traditional web application approach where each page load results in a new request to the server, which then returns a full HTML page. MPAs are more suited for larger applications with multiple categories or sections.

Characteristics of MPAs:

- **Multiple HTML Files**: Each page is represented by a separate HTML file that the server sends to the client.
- **Server-side Rendering**: Content is generated on the server and sent to the client as fully rendered HTML.
- **Full Page Reloads**: Every time a user navigates to a different page, the browser reloads the entire page.
- **SEO-friendly**: Since each page is a separate HTML file, MPAs are generally easier to optimize for search engines.
- **Examples**: E-commerce websites, blogs, news portals.

Technologies Commonly Used in MPAs:

- **Server-side Frameworks**:
    - **Laravel (PHP)**, **Django (Python)**, **Ruby on Rails (Ruby)**, **Spring (Java)**: These frameworks handle server-side logic, rendering views, and serving HTML files.
- **Templating Engines**:
    - **Blade (Laravel)**, **Jinja2 (Django)**, **ERB (Rails)**: These engines allow the dynamic generation of HTML based on server-side data.
- **Frontend Enhancements**:
    - **jQuery**: Often used for adding interactive features without building a full SPA.

- **AJAX**: Used to asynchronously load data without reloading the entire page.

Advantages of MPAs:

- **SEO Benefits**: Each page is a separate HTML document, making them easier to crawl and index by search engines.
- **Simplicity**: Easier to implement and understand, particularly for small to medium-sized applications.
- **Scalability**: MPAs can easily scale by adding more pages.

Disadvantages of MPAs:

- **Slower User Experience**: Full-page reloads can lead to a less responsive user experience compared to SPAs.
- **Higher Server Load**: Each page request results in a full round trip to the server, increasing server load and bandwidth usage.
- **Complexity in State Management**: Maintaining a consistent user experience across pages can be challenging.

Different Client-side Technologies

1. JavaScript Libraries and Frameworks

- **React.js**: A powerful library for building dynamic UIs, particularly suited for SPAs. It uses a component-based architecture and a virtual DOM for efficient updates.
- **Angular**: A full-fledged framework for building complex SPAs. It includes tools for routing, form handling, and state management.
- **Vue.js**: A flexible and lightweight framework that can be incrementally adopted in projects. It's popular for its ease of integration and progressive nature.

2. AJAX (Asynchronous JavaScript and XML)

- A technique for creating asynchronous web applications by exchanging data with a server and updating parts of a web page without reloading the entire page.

3. WebAssembly

- A low-level bytecode for the web, WebAssembly enables code written in multiple languages (like C, C++, Rust) to run on the web with near-native performance.

4. Web Components

- A set of standards for creating reusable custom elements in web applications. Web components encapsulate HTML, CSS, and JavaScript, allowing developers to create modular and reusable UI components.

5. Progressive Web Apps (PWAs)

- A type of web application that leverages modern web capabilities to deliver an app-like experience. PWAs can work offline, send push notifications, and be installed on the user's home screen.

6. TypeScript

- A typed superset of JavaScript that compiles to plain JavaScript. TypeScript is used to catch errors at compile-time and improve code maintainability, especially in large projects.

7. CSS Preprocessors and Frameworks

- **Sass, LESS**: Preprocessors that extend CSS with variables, nesting, and functions, making it easier to maintain large stylesheets.
- **Bootstrap, Tailwind CSS**: CSS frameworks that provide pre-designed components and utility classes for faster and consistent UI development.

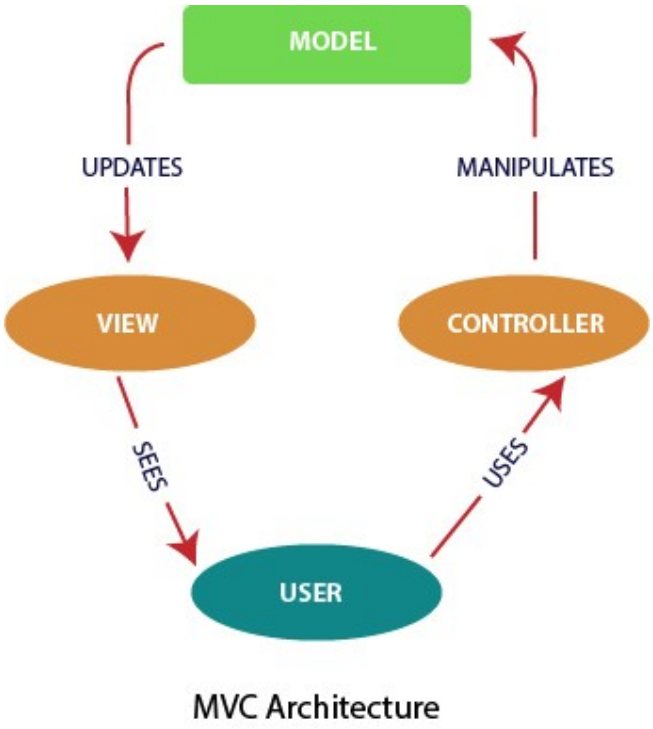
8. Frontend Build Tools

- **Webpack, Parcel, Vite**: Tools that bundle and optimize assets like JavaScript, CSS, and images for production, improving load times and performance.
- **Babel**: A JavaScript compiler that allows developers to use the latest JavaScript features by transpiling code to be compatible with older browsers.

These client-side technologies, combined with the appropriate architectural approach (SPA or MPA), are essential for building modern, responsive, and efficient web applications.

1.1.2 MVC Architecture

In React, the concept of MVC (Model-View-Controller) architecture is not directly enforced, but you can still apply the principles of MVC to structure your React application in a way that aligns with this design pattern. Here's how the MVC architecture can be mapped to a React-based application:



MVC Architecture

1. View (React Components)

In React, the **View** is represented by the React components. React components are responsible for rendering the UI and managing the visual aspects of the application.

- **Responsibilities**:
    - **Rendering UI**: React components render the user interface based on the data they receive via props or state.
    - **Handling User Interaction**: Components can capture user input, such as clicks, form submissions, and other events. They can also manage local state using hooks like useState.
    - **Presentational Logic**: React components handle how the data should be displayed, applying styles, formatting, and other presentational logic.
- **Examples**:
    - A PostList component might render a list of blog posts.
    - A ProductDetail component might display the details of a specific product, such as its name, price, and description.

2. Controller (React Components & Event Handlers)

In the React context, the **Controller** role is typically handled within the React components themselves, especially through event handlers and lifecycle methods

(or hooks). The controller logic is embedded in the components to manage user interactions and update the state accordingly.

- **Responsibilities**:
    - **Handling User Actions**: Components manage user actions, such as button clicks, form submissions, or input changes, often triggering state updates or API calls.
    - **State Management**: The controller logic, embedded within components, uses React's state management (useState, useReducer) to update the UI in response to user actions.
    - **Interacting with the Model**: React components may call functions that interact with the Model (e.g., making API calls to fetch or update data) and then update the component state based on the results.
- **Examples**:
    - A handleSubmit function in a Form component might process form data and update the state or make an API call to save the data.
    - An onClick event handler in a button might trigger a state update that re-renders part of the UI.

3. Model (State Management & APIs)

In a React application, the **Model** is typically represented by the state of the application, which can be managed in various ways. The Model might also involve interaction with external APIs or databases to fetch or persist data.

- **Responsibilities**:
    - **Managing Application State**: The Model's data is managed using React's state (useState, useReducer), context (useContext), or external state management libraries like Redux or MobX.
    - **Data Logic and Business Rules**: The Model handles the logic for processing data, such as validation, calculations, and other business rules before it is passed to the View.
    - **Interacting with Data Sources**: The Model interacts with APIs or databases to fetch or update data, often through functions that are called from within components.
- **Examples**:
    - A useState hook in a component might hold the data for a form.
    - A fetchPosts function might retrieve a list of blog posts from an API and update the state accordingly.

- A Redux store might manage the global state of the application, handling data and business logic that multiple components rely on.

Putting It All Together: MVC in React

1. **Model**: The state of your application, managed through React's state hooks, context, or a state management library like Redux, represents the Model. This includes data fetched from APIs, user input, and any business logic.
2. **View**: React components are your View. They take the state (Model) and render it into the UI. They are responsible for presenting data to the user and capturing user interactions.
3. **Controller**: In React, the controller logic is usually embedded within the components through event handlers and lifecycle methods. This logic manages user input, updates the Model, and triggers re-renders of the View.

Example: A Simple React Application Using MVC Principles

Let's consider a simple React app that displays a list of tasks (a to-do list):

- **Model**:
  - 
    ```
    const [tasks, setTasks] = useState([]); // This is the Model storing tasks
    ```
  - The tasks state holds the list of tasks. You might fetch tasks from an API and update this state.
- **View**:
  - 
    ```
    function TaskList({ tasks }) {
      return (
        <ul>
          {tasks.map((task, index) => (
            <li key={index}>{task}</li>
          ))}
        </ul>
      );
    }
    ```
  - The TaskList component takes the tasks (Model) as props and renders them as a list.
- **Controller**:

```
function App() {
  const [tasks, setTasks] = useState([]);

  const addTask = (task) => {
    setTasks([...tasks, task]); // Updates the Model
  };

  return (
    <div>
      <TaskList tasks={tasks} />
      <AddTaskForm onAddTask={addTask} /> {/* Controller logic handles adding a task */}
    </div>
  );
}
```

- The App component manages the state (Model) and handles user interactions, such as adding a new task through the addTask function. It passes data and functions down to other components.

In this way, while React does not enforce MVC, you can structure your React application to follow MVC principles, with clear separation between data management (Model), UI rendering (View), and interaction logic (Controller).

1.1.3 Introduction to React, Installation of React JSX and its use case

What is React?

React is a JavaScript library for building user interfaces. It's a component-based framework, meaning it breaks down complex UIs into smaller, reusable pieces called components. This modular approach makes it easier to manage and maintain large-scale applications.

What is JSX?

JSX is a syntax extension for JavaScript that allows you to write HTML-like code within JavaScript files. It's not technically JavaScript but is transformed into regular JavaScript code before it's executed. JSX makes it easier to write and understand React components.

Why Use React and JSX?

- **Component-Based Architecture:** Encourages reusability and modularity.
- **Declarative Syntax:** You describe what you want the UI to look like, and React handles the details of how to render it.
- **Virtual DOM:** React uses a virtual DOM, which is a lightweight representation of the actual DOM. This allows React to efficiently update the UI by only changing the necessary parts.

- **Large Community and Ecosystem:** A vast community of developers contributes to React, and there are numerous third-party libraries and tools available to enhance your development experience.

Installing React and JSX

1. **Install Node.js:** Ensure you have Node.js and npm (Node Package Manager) installed on your system. You can download it from https://nodejs.org/en/download/package-manager.
2. **Create a New Project:** Use npm to create a new React project:

```
npx create-react-app my-react-app
```

3. **Navigate to the Project Directory:**

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

4. **Start the Development Server:**

```
npm start
```

This will start a development server and open your React application in your default browser.

Using JSX in React Components

Here's a simple example of a React component using JSX:

```jsx
import React from 'react';

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

export default Greeting;
```

In this example, Greeting is a functional component that takes a name prop. The JSX syntax is used to define the structure and content of the component's UI. The props.name is interpolated within the <h1> element to display the greeting message.

Key Points:

- JSX is compiled into JavaScript.
- You can use JSX to define HTML-like elements and attributes.
- Props are passed to components to customize their behavior.
- React components can be nested within each other to create complex UIs.

By understanding these concepts, you're well-equipped to start building React applications using JSX.

1.1.4 DOM, Virtual DOM and its working

DOM (Document Object Model)

The DOM is a programming interface that represents an HTML document as a tree structure. It provides a way for JavaScript to interact with the elements of a web page. Each element in the DOM is represented as a node, and these nodes have properties and methods that can be used to manipulate the document.

Key Points:

- The DOM is a tree structure.
- Each element in the DOM is a node.
- JavaScript can access and manipulate the DOM using properties and methods.

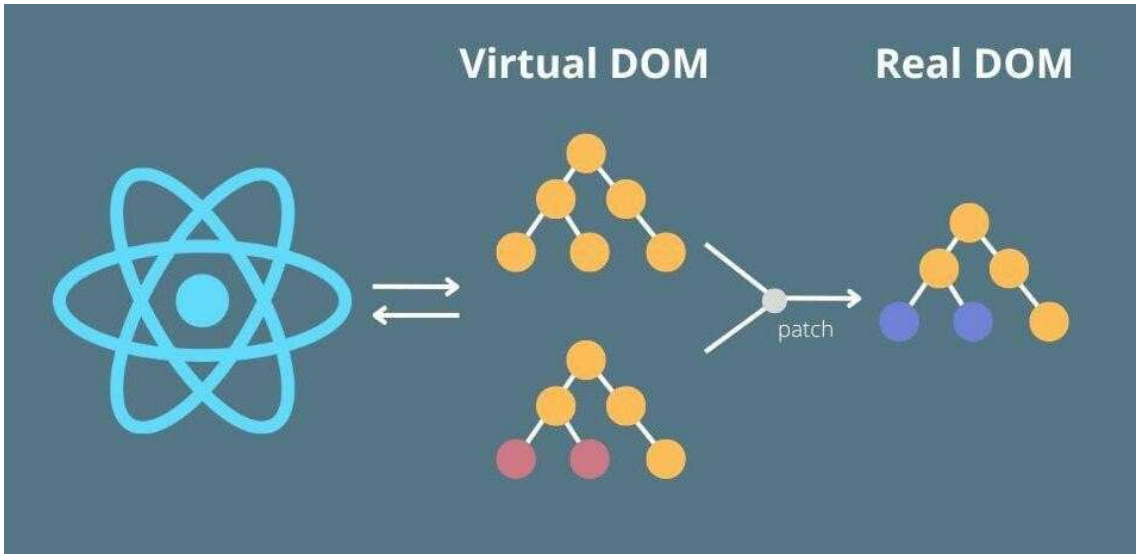
Virtual DOM

The virtual DOM is a lightweight, in-memory representation of the actual DOM. It's a copy of the DOM that React creates and updates before making changes to the actual DOM. This approach is more efficient than directly manipulating the DOM, especially for complex applications with frequent updates.

Key Points:

- The virtual DOM is a copy of the actual DOM.

- React creates and updates the virtual DOM before making changes to the actual DOM.
- The virtual DOM is more efficient than directly manipulating the DOM.



How the Virtual DOM Works

1. **Initial Rendering:** When a React component is rendered for the first time, React creates a virtual DOM representation of the component's UI.
2. **State or Prop Changes:** When a component's state or props change, React re-renders the component, creating a new virtual DOM.
3. **Diffing:** React compares the new virtual DOM with the previous one to identify the differences.
4. **Updating the Actual DOM:** React updates only the necessary parts of the actual DOM based on the identified differences.

Example: Suppose you have a React component that renders a list of items. When a new item is added to the list, React does the following:

1. Creates a new virtual DOM with the added item.
2. Compares the new virtual DOM with the previous one.
3. Identifies that only the list element needs to be updated.
4. Updates the list element in the actual DOM to reflect the new item.

By using the virtual DOM, React can efficiently update the UI without having to re-render the entire page. This can significantly improve performance, especially for large and complex applications.

1.1.5 ECMAScript, Difference between ES5 and ES6.

What is ECMAScript?

ECMAScript (ES) is a scripting language specification upon which JavaScript is based. It standardizes how JavaScript is implemented, providing a foundation for writing and running JavaScript code across different environments (browsers, servers, etc.). ECMAScript versions are often referred to as ES followed by the version number, such as ES5, ES6, etc.

ES5 (ECMAScript 5)

ECMAScript 5 was released in 2009 and became widely adopted as the de facto standard for JavaScript for several years. ES5 introduced several new features that made JavaScript more powerful and easier to work with.

Key Features of ES5:

1. **Strict Mode**:
    - Introduced a way to opt-in to a restricted variant of JavaScript, providing better error-checking and avoiding common pitfalls.

```
1 'use strict';
2 x = 3.14; // Throws an error because x is not declared
```

2. **Array Methods**:
    - New array methods were introduced, such as forEach(), map(), filter(), reduce(), and some(), which provide functional ways to manipulate arrays.

```
1 var numbers = [1, 2, 3];
2 var doubled = numbers.map(function(num) {
3   return num * 2;
4 });
5 console.log(doubled); // Output: [2, 4, 6]
```

3. **JSON Support**:

- ES5 provided built-in support for parsing and stringifying JSON data with JSON.parse() and JSON.stringify().

```
var jsonString = '{"name":"John", "age":30}';
var obj = JSON.parse(jsonString);
console.log(obj.name); // Output: John
```

4. **Object Methods**:
   - New methods like Object.create(), Object.defineProperty(), and Object.keys() were added for more control over object properties and their descriptors.

```
var person = {
  name: 'John',
  age: 30
};
var keys = Object.keys(person);
console.log(keys); // Output: ["name", "age"]
```

   - **Property Getters and Setters**:
   - ES5 introduced getter and setter functions for managing object property values dynamically.

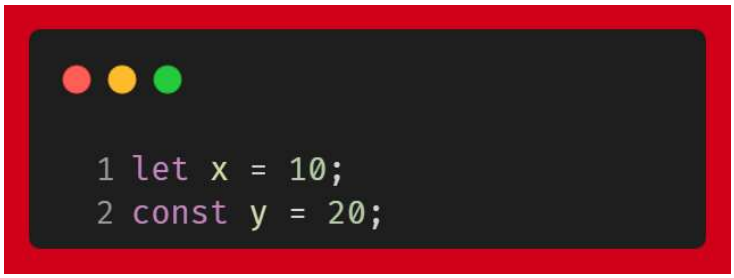
```
var person = {
  firstName: "John",
  lastName: "Doe",
  get fullName() {
    return this.firstName + " " + this.lastName;
  }
};
console.log(person.fullName); // Output: John Doe
```

ES6 (ECMAScript 6) / ES2015

ECMAScript 6 (ES6), also known as ECMAScript 2015, was a significant update to the language and introduced many new features that modernized JavaScript and made it more powerful. ES6 is widely used in modern JavaScript development.

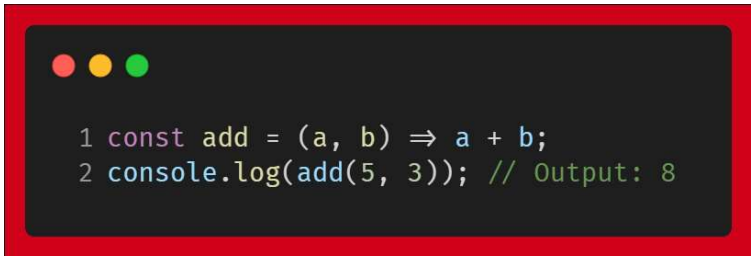
Key Features of ES6:

1. **let and const**:
    - let and const were introduced as new ways to declare variables, providing block-scoping and preventing variable hoisting issues.
    - let is used for variables that may change, while const is used for variables that should remain constant.



```
1 let x = 10;
2 const y = 20;
```

2. **Arrow Functions**:
    - Arrow functions provide a shorter syntax for writing functions and automatically bind the this value from the surrounding context.

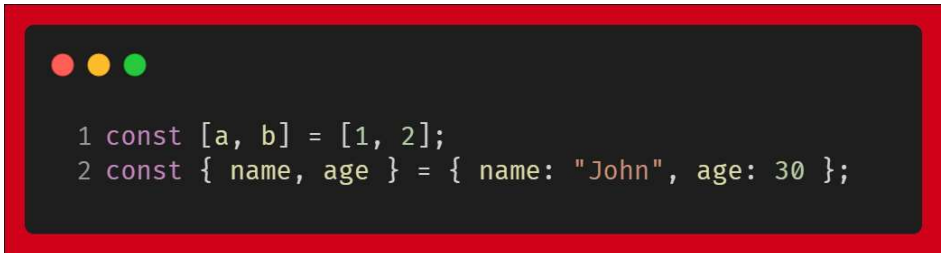


```
1 const add = (a, b) ⇒ a + b;
2 console.log(add(5, 3)); // Output: 8
```

3. **Template Literals**:
    - Template literals allow for easier string interpolation and multi-line strings, improving readability and flexibility.

```
1 const name = "John";
2 const greeting = `Hello, ${name}!`;
3 console.log(greeting); // Output: Hello, John!
```

4. **Destructuring Assignment**:
    - Destructuring allows for unpacking values from arrays or properties from objects into distinct variables.

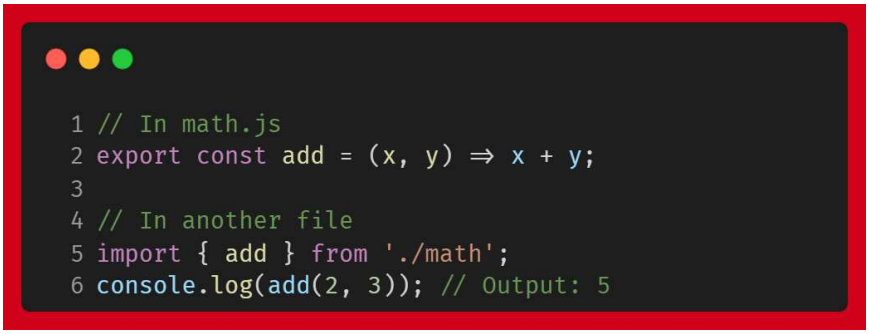


```
1 const [a, b] = [1, 2];
2 const { name, age } = { name: "John", age: 30 };
```

5. **Classes**:
    - ES6 introduced classes as syntactical sugar over JavaScript's existing prototype-based inheritance, making it easier to work with OOP principles.

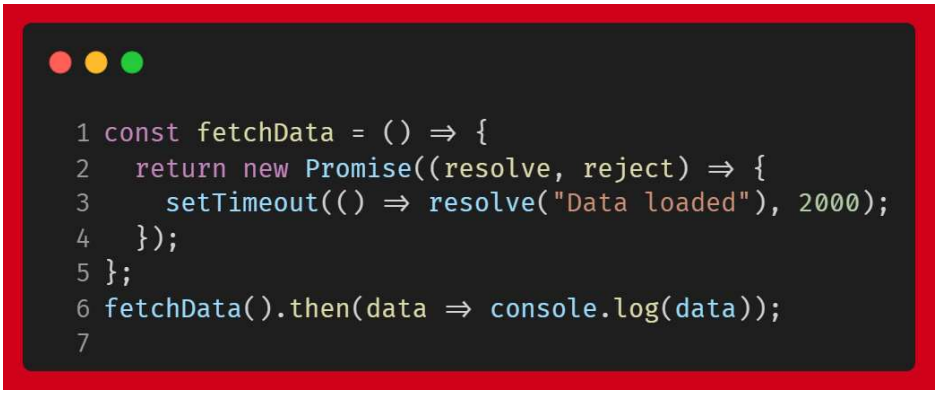
```
1 class Person {
2   constructor(name, age) {
3     this.name = name;
4     this.age = age;
5   }
6   greet() {
7     console.log(`Hello, my name is ${this.name}`);
8   }
9 }
10 const john = new Person('John', 30);
11 john.greet(); // Output: Hello, my name is John
```

6. **Modules**:
    - ES6 introduced native module support with the import and export keywords, allowing for better code organization and reuse.



```
// In math.js
export const add = (x, y) => x + y;

// In another file
import { add } from './math';
console.log(add(2, 3)); // Output: 5
```

7. **Promises**:
    - Promises provide a more manageable way to handle asynchronous operations, avoiding callback hell and improving code readability.



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

```

8. **Default Parameters**:
    - Functions in ES6 can have default parameter values, reducing the need for manual checks and assignments within function bodies.

```
function greet(name = "Guest") {
  console.log(`Hello, ${name}!`);
}
greet(); // Output: Hello, Guest!
```

9. **Spread and Rest Operators**:

- The spread (...) and rest (...) operators provide a concise way to expand or collect elements.

```
1 const arr = [1, 2, 3];
2 const newArr = [ ...arr, 4, 5];
3 console.log(newArr); // Output: [1, 2, 3, 4, 5]
```

Comparison: ES5 vs. ES6

Feature	ES5	ES6
Variable Declaration	var (function-scoped)	let and const (block-scoped)
Functions	Traditional function expressions	Arrow functions with lexical this
String Handling	Concatenation using + operator	Template literals with ${} for interpolation
Object Properties	Manual assignment in constructors	Shorthand syntax and destructuring assignment
Inheritance	Prototype-based inheritance with manual setup	Class syntax with constructor and extends
Modules	No native support (relied on CommonJS, AMD)	Native module system with import and export
Asynchronous Handling	Callback functions, manual error handling	Promises and async/await (introduced in ES8)
Loops	for, while, do-while, forEach	Addition of for...of loop for iterables
Default Parameters	Manual checks for undefined	Default parameter values

Arrays and Objects	Limited built-in methods for manipulation	Spread/rest operators, new array methods (find, findIndex, etc.)

Conclusion

ES6 introduced many features that simplified and modernized JavaScript development. While ES5 laid the foundation, ES6 brought JavaScript closer to other modern programming languages in terms of functionality and ease of use. Understanding the differences between ES5 and ES6 is essential for working effectively with JavaScript, especially in modern web development.

1.1.6 NPM Modules

NPM (Node Package Manager) is a package manager for JavaScript that allows you to easily install and manage third-party libraries and tools. In React development, NPM is essential for incorporating additional functionalities and streamlining the development process.

Common NPM Modules Used in React

Here are some popular NPM modules commonly used in React projects:

- **State Management:**
  - **Redux:** A predictable state container for JavaScript applications.
  - **Context API:** A built-in React feature for sharing data between components without prop drilling.
  - **MobX:** A simple, scalable state management solution.
- **Styling:**
  - **Styled Components:** A CSS-in-JS library for styling React components.
  - **Sass/Less:** CSS preprocessors for writing more maintainable stylesheets.
- **Routing:**
  - **React Router:** A popular routing library for creating single-page applications.
- **Testing:**
  - **Jest:** A JavaScript testing framework.

    - **Enzyme:** A testing utility for React.
- **Form Handling:**
    - **Formik:** A higher-order component for managing forms in React.
    - **Yup:** A JavaScript schema builder for validation.
- **Data Fetching:**
    - **Axios:** A promise-based HTTP client for making requests to REST APIs.
    - **SWR:** A React hook for fetching data from a remote API.
- **UI Components:**
    - **Material-UI:** A popular UI component library based on Google's Material Design.
    - **Ant Design:** A design system for enterprise-level applications.
    - **Chakra UI:** A modular and accessible component library.
- **Miscellaneous:**
    - **Lodash:** A utility library for JavaScript.
    - **Moment.js:** A JavaScript library for manipulating dates and times.
    - **React Helmet:** A library for managing document head elements in React.

Installing NPM Modules

To install an NPM module, use the npm install command followed by the module name:

$> npm install react-router-dom

This will install the react-router-dom module and add it to your package.json file.

Using NPM Modules in React

Once a module is installed, you can import it into your React components and use its functions and components. For example, to use react-router-dom for routing:

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

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/about" element={<AboutPage    />} />
      </Routes>
    </Router>
  );
}
```

By leveraging NPM modules, you can significantly enhance the functionality and development experience of your React applications.

1.2 React Elements:

React elements are the building blocks of React applications. They are the smallest units of a React app and are responsible for describing what you want to see on the screen.

1.2.1 Render Function, Components, Class Component, Component Constructor, Functional Components, Multiple Components,

1.2.1.1 Render Function

The render function is a crucial part of React. It defines the output of a React component, specifying what should be displayed in the DOM.

- **In React Class Components**: The render method is mandatory and must return a React element (typically JSX).

Example:

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

- **In Functional Components**: The function itself acts as the render method and returns the JSX to be rendered.

Example:

```
function MyComponent() {
  return <h1>Hello, World!</h1>;
}
```

1.2.1.2 Components

Components are the core units in React that allow you to split the UI into independent, reusable pieces. They can be classified into two types:

1. **Class Components**
2. **Functional Components**

Example of a Component:

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

1.2.1.3 Class Components

Class Components are ES6 classes that extend from React.Component. They can have their own state and lifecycle methods.

Example:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- **Component Constructor**: In a class component, the constructor is used to initialize state and bind methods. It is called when the component is created.

Example:

```
class Welcome extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return <h1>Count: {this.state.count}</h1>;
  }
}
```

- The super(props) call is necessary to pass props to the parent React.Component class.

1.2.1.4 Functional Components

Functional Components are JavaScript functions that accept props as an argument and return a React element. They are simpler than class components and are preferred in modern React development due to their simplicity and performance.

Example:

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

- **Advantages**:
  - Easier to write and understand.
  - No need for this keyword.
  - Better performance since they are simpler and have less overhead.
  - Can utilize React hooks for state and side effects.

1.2.1.5 Multiple Components

React allows you to create complex UIs by composing multiple components. Each component can manage its own state and props, making it easy to build and maintain large applications.

Example:

```
function Header() {
  return <header><h1>My App</h1></header>;
}

function Footer() {
  return <footer><p>© 2024 My App</p></footer>;
}

function App() {
  return (
    <div>
      <Header />
      <main>
        <p>Welcome to my app!</p>
      </main>
      <Footer />
    </div>
 );
}
```

- In this example, the App component is composed of Header, Footer, and a main content area. Each part of the UI is encapsulated in its own component, promoting reusability and clarity.

Conclusion

React components, whether class-based or functional, are the fundamental units that drive the architecture of React applications. They encapsulate rendering logic, handle state, and allow you to break down the UI into reusable, manageable pieces. Understanding how to effectively use components, along with the render function and constructors, is essential for building robust React applications.

1.2.2 Props: Props with Class based Component, Props with Function based Component, States,

Props and **state** are two essential concepts in React that help manage data flow and component behavior. Understanding how they work with both class-based and functional components is crucial for developing React applications.

1.2.2.1 Props

Props (short for "properties") are read-only data passed from a parent component to a child component. They allow components to be dynamic and reusable by enabling them to receive different data inputs.

Props with Class-based Components

In class components, props are accessed using this.props.

Example:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}

// Usage
<Welcome name="Alice" />
```

- In this example, the Welcome component receives a name prop, which it then uses to render a greeting.

Props with Function-based Components

In functional components, props are passed as arguments to the function.

Example:

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

// Usage
<Welcome name="Bob" />
```

- Here, the Welcome function receives props as an argument and uses props.name to render the greeting.

Destructuring Props:

- You can destructure props directly in the function signature for cleaner and more readable code.

Example:

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

1.2.2.2 State

State is a built-in object that allows components to hold and manage their own data. Unlike props, state is mutable, meaning it can be updated over time. State is typically used to handle data that changes as the user interacts with the app.

State in Class-based Components

In class components, state is initialized in the constructor and updated using this.setState().

Example:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

// Usag
<Counter />
```

- In this example, the Counter component has a count state that starts at 0. The increment method updates the state using this.setState, and the component re-renders with the updated count.

State in Function-based Components

In functional components, state is managed using the useState hook.

Example:

```
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>
  );
}

// Usage
<Counter />
```

- The useState hook returns an array with two elements: the current state value (count) and a function to update it (setCount).
  The useState(0) initializes the count state to 0.

Updating State:

- State updates in both class and functional components are asynchronous. React batches state updates to improve performance, so multiple updates may be merged into a single render.

Comparison: Props vs. State

Aspect	Props	State
Data Source	Passed from parent components	Managed internally within the component
Mutability	Immutable (read-only)	Mutable (can be updated)
Responsibility	Makes components reusable by receiving	Manages data that changes over time due to user interaction

	dynamic data	
Usage	Ideal for passing data from parent to child components	Ideal for managing component-specific data that changes
Access in Class	this.props	this.state
Access in Function	Passed as function arguments (props)	Managed using the useState hook

---

Conclusion

Understanding the roles of props and state in React is fundamental for building interactive UIs. Props allow for data to flow into components, making them dynamic and reusable, while state enables components to manage and respond to changes over time. Whether you're working with class-based or function-based components, mastering props and state will give you the tools you need to build robust React applications.

1.2.3 Component Lifecycle

React components have a lifecycle, which refers to the various stages they go through during their existence. Understanding the component lifecycle is essential for managing state, side effects, and optimizing performance.

Key Lifecycle Methods

- **constructor(props):** Called when the component is created. Use it to initialize state or bind methods.
- **componentDidMount():** Called after the component is mounted into the DOM. Use it for fetching data, setting up subscriptions, or performing side effects.
- **componentDidUpdate(prevProps, prevState):** Called when the component's props or state have changed. Use it to update the DOM or perform side effects based on the changes.
- **componentWillUnmount():** Called before the component is unmounted from the DOM. Use it to clean up subscriptions, timers, or other resources.

Lifecycle Diagram

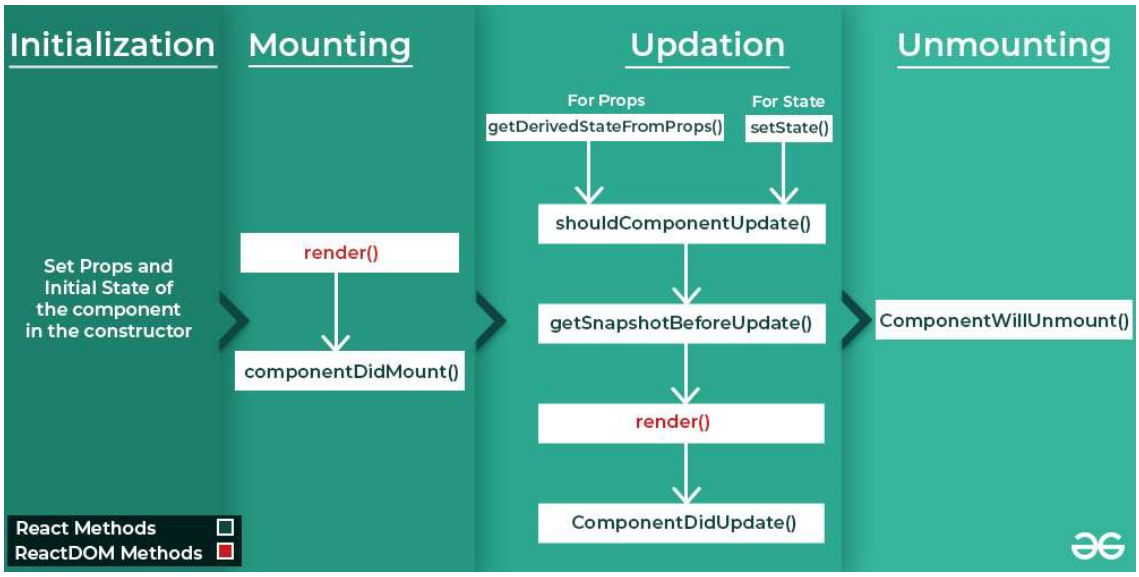
Example

```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount()    {
    // Fetch data from an API
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
        this.setState({ data });
      });
  }

  componentDidUpdate(prevProps, prevState) {
    if    (prevState.count !== this.state.count) {
      // Update the DOM based on the count change
      console.log('Count has changed!');
    }
  }

  componentWillUnmount() {
    // Clean up any subscriptions or timers
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}
```

In this example:

- The constructor initializes the component's state.
- componentDidMount fetches data from an API and updates the state.
- componentDidUpdate logs a message when the count state changes.
- componentWillUnmount is not used in this example but could be used to clean up any subscriptions or timers.

By understanding the component lifecycle, you can effectively manage the behavior and state of your React components.



1.3 React Events

React provides a way to handle user interactions with your components through events. Events are JavaScript objects that contain information about the event, such as the target element and the event type.

Common Event Types

- **onClick:** Triggered when an element is clicked.
- **onMouseOver:** Triggered when the mouse pointer enters an element.
- **onMouseOut:** Triggered when the mouse pointer leaves an element.
- **onChange:** Triggered when an input element's value changes.
- **onSubmit:** Triggered when a form is submitted.
- **onKeyDown:** Triggered when a key is pressed down.
- **onKeyUp:** Triggered when a key is released.

Handling Events in React

To handle an event in React, you can attach an event handler as a prop to the element. The event handler is a function that will be called when the event occurs.

```
function MyComponent() {
  const handleClick = () => {
    console.log('Button clicked!');
  };

  return (
    <button onClick={handleClick}>Click me</button>
  )
}
```

In this example, the handleClick function is attached to the onClick prop of the button element. When the button is clicked, the handleClick function will be called.

Passing Event Data

You can access the event object within the event handler function. This allows you to get information about the event, such as the target element or the event type.

```
function MyComponent() {
  const handleInputChange = (event) => {
    console.log(event.target.value);
  };

  return (
    <input type="text" onChange={handleInputChange} />
  );
}
```

In this example, the handleInputChange function is attached to the onChange prop of the input element. When the input value changes, the function will be called and the event.target.value property will contain the new value.

Preventing Default Behavior

Sometimes you may want to prevent the default behavior of an event. For example, you might want to prevent a form from submitting when a button is clicked. You can do this using the preventDefault method of the event object.

```jsx
function MyComponent() {
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log('Form submitted!');
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" name="username" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

By calling event.preventDefault() in the handleSubmit function, we prevent the form from submitting and instead log a message to the console.

By effectively handling events, you can create interactive and responsive React components.

1.4 React Forms, Different Form Concepts.

React forms are used to collect user input and submit data to the server. They are essential for building interactive web applications.

Basic Form Structure

A basic React form consists of the following elements:

- **<form>:** The root element that defines the form.
- **<input>:** Used to collect various types of input, such as text, numbers, checkboxes, and radio buttons.
- **<select>:** Used to create dropdown lists.
- **<textarea>:** Used to create multi-line text areas.
- **<button>:** Used to submit the form or perform other actions.

Handling Form Submission

To handle form submission, you can attach an onSubmit event handler to the <form> element. The event handler will be called when the form is submitted.

```
function MyForm() {
  const handleSubmit = (event) => {
    event.preventDefault();
    // Handle form submission here
    console.log(event.target.elements.username.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>Username:</label>
      <input type="text" name="username" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

In this example, the handleSubmit function is called when the form is submitted. We prevent the default behavior of the form using event.preventDefault() and then access the value of the username input field.

Controlled Components

Controlled components are components where the value of an input element is managed by the component's state. This allows you to have more control over the input and validate it before submitting the form.

```
function MyForm() {
  const [username, setUsername] = useState('');

  const handleSubmit = (event) => {
    event.preventDefault();
    // Handle form submission here
    console.log(username);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>Username:</label>
      <input type="text" value={username} onChange={(event) =>
setUsername(event.target.value)} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

In this example, the username state is used to control the value of the input element. When the input value changes, the setUsername function is called to update the state.

Uncontrolled Components

Uncontrolled components are components where the value of an input element is managed by the DOM. This is a simpler approach but can be less flexible.

```
function MyForm() {
  const handleSubmit = (event) => {
    event.preventDefault();
    // Handle form submission here
    console.log(event.target.elements.username.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>Username:</label>
      <input type="text" name="username" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

In this example, the value of the username input is managed by the DOM. We access it directly in the handleSubmit function.

Form Validation

You can validate form input to ensure that it meets certain criteria. This can be done using regular expressions, custom validation functions, or third-party libraries.

```
function MyForm() {
  const [username, setUsername] = useState('');
  const [errors, setErrors] = useState({});

  const handleSubmit = (event) => {
    event.preventDefault();
    // Validate form here
    const newErrors = {};
    if (!username) {
      newErrors.username = 'Username is required';
    }
    setErrors(newErrors);
    if (Object.keys(newErrors).length === 0) {
      // Submit form
      console.log(username);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>Username:</label>
      <input type="text" value={username} onChange={(event) =>
setUsername(event.target.value)}    />
      {errors.username && <p>{errors.username}</p>}
      <button type="submit">Submit</button>
    </form>
  );
}
```

In this example, we validate the username input to ensure that it is not empty. We display an error message if the validation fails.

By understanding these concepts, you can effectively create and manage forms in your React applications.

1.5 Styling in React and Inline Styling

Styling in React allows you to create visually appealing and responsive user interfaces. React supports various approaches for styling components, including inline styles, CSS stylesheets, CSS modules, and styled-components. Each method has its benefits and use cases.

1.5.1 Inline Styling

Inline Styling involves applying CSS styles directly within a React component using JavaScript objects. This approach provides a quick way to apply styles but can become unwieldy for complex styles.

Key Points:

- Styles are defined as JavaScript objects.
- Inline styles are applied directly to the style attribute of HTML elements.
- The style object properties are written in camelCase rather than kebab-case.

Example:

```
function InlineStyledComponent() {
  const divStyle = {
    color: 'blue',
    backgroundColor: 'lightgray',
    padding: '10px',
    borderRadius: '5px',
  };

  return (
    <div style={divStyle}>
      This is an inline styled div.
    </div>
  );
}
```

- **divStyle**: A JavaScript object representing CSS styles.
- **style={divStyle}**: The style is applied directly to the div element.

Advantages:

- Simple to implement for small or dynamic styles.
- Scoped to the component, preventing styles from leaking.

Disadvantages:

- Can become cumbersome for complex or large styles.
- Lacks pseudo-classes (like :hover) and media queries support.

1.5.2 Using CSS Stylesheets

CSS Stylesheets involve using traditional .css files to define styles and importing them into your React components. This approach leverages standard CSS and is well-suited for larger applications.

Example:

Create a CSS File (e.g., App.css):

```css
.container {
  color: blue;
  background-color: lightgray;
  padding: 10px;
  border-radius: 5px;
}
```

Import and Use the CSS File:

```jsx
import React from 'react';
import './App.css';

function StyledComponent() {
  return (
    <div className="container">
      This is styled using a CSS stylesheet.
    </div>
  );
}

```

- **import './App.css'**: Imports the CSS file into the component.
- **className="container"**: Applies the CSS class to the div element.

Advantages:

- Standard approach for styling that integrates with existing CSS practices.
- Supports all CSS features, including pseudo-classes and media queries.

Disadvantages:

- Global scope can lead to style conflicts.
- Requires manual management of class names and potential naming collisions.

1.5.3 CSS Modules

CSS Modules provide a way to scope CSS by automatically generating unique class names. This approach helps prevent global namespace collisions and makes it easier to manage styles.

Example:

Create a CSS Module File (e.g., App.module.css):

```css
.container {
  color: blue;
  background-color: lightgray;
  padding: 10px;
  border-radius: 5px;
}
```

Import and Use the CSS Module:

```jsx
import React from 'react';
import styles from './App.module.css';

function StyledComponent() {
  return (
    <div className={styles.container}>
      This is styled using CSS Modules.
    </div>
  );
}
```

- **import styles from './App.module.css'**: Imports CSS module styles as an object.
- **className={styles.container}**: Applies the scoped class name.

Advantages:

- Scoped styles prevent conflicts and naming collisions.
- Maintains the power of CSS with local scoping.

Disadvantages:

- Requires build tools (like Webpack) that support CSS Modules.
- Slightly more complex setup compared to global CSS.

1.5.4 Styled Components

Styled Components is a library for styling React components using tagged template literals. It allows for writing actual CSS to style components while leveraging JavaScript.

Example:

Install Styled Components:

npm install styled-components

Use Styled Components:

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

const StyledDiv = styled.div`
  color: blue;
  background-color: lightgray;
  padding: 10px;
  border-radius: 5px;
`;

function StyledComponent() {
  return (
    <StyledDiv>
      This is styled using Styled Components.
    </StyledDiv>
  );
}
```

- **styled.div**: Creates a styled div component with the specified CSS.
- **Template literals**: Write CSS directly within JavaScript.

Advantages:

- Component-scoped styles with dynamic capabilities.
- Supports themes, nesting, and more advanced CSS features.

Disadvantages:

- Adds a dependency to your project.
- May introduce some performance overhead compared to traditional CSS.

Conclusion

React provides various methods for styling components, each with its strengths and use cases. Inline styling is quick and simple for small tasks, while CSS stylesheets and CSS modules offer more traditional and scalable approaches. Styled Components offer a modern, component-scoped styling solution with advanced features. Choosing the right approach depends on your project's needs, team preferences, and the complexity of your styles.

[Practical Applications of Unit-1: Build Music shop application using Unit-1]

Unit 2.

2.1 Routing: react-router, Features of react-router, Configuration of routing using react-router

React Router: A Guide to Routing in React

React Router is a popular library for creating single-page applications (SPAs) in React. It provides a declarative way to manage navigation and URL changes within your application.

Key Features of React Router

- **Declarative Routing:** You define routes using a simple declarative syntax, making it easy to manage navigation.
- **Nested Routes:** Create complex routing structures by nesting routes within each other.
- **Dynamic Routes:** Use path parameters to create dynamic routes that can handle different URL patterns.
- **Programmatic Navigation:** Navigate to different routes programmatically using the useNavigate hook or the history object.
- **Route Protection:** Protect routes using authentication or authorization mechanisms.

Configuration of Routing Using React Router

To use React Router in your project, you'll need to install it using npm or yarn:

$> npm install react-router-dom

Then, import the necessary components from react-router-dom and configure the routing in your main app component:

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

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/about" element={<AboutPage />} />
        <Route path="/contact" element={<ContactPage />} />
      </Routes>
    </Router>
  );
}
```

In this example:

- BrowserRouter provides the routing context for the application.
- Routes defines the collection of routes.
- Route defines a specific route, including the path and the component to render.

Dynamic Routes

To create dynamic routes, use path parameters enclosed in colons:

```jsx
<Route path="/users/:userId" element={<UserProfile />} />
```

In this example, the userId parameter will be extracted from the URL and passed as a prop to the UserProfile component.

Nested Routes

To create nested routes, nest Route components within other Route components:

```
<Route path="/products" element={<ProductsPage />}>
  <Route path="/products/:productId" element=
 {<ProductDetails />} />
</Route>
```

This creates a nested route for /products/:productId within the /products route.

Programmatic Navigation

To navigate to different routes programmatically, use the useNavigate hook:

```
import { useNavigate } from 'react-router-dom';

function MyComponent() {
  const navigate = useNavigate();

  const handleClick = () => {
    navigate('/about');
  };

  return    (
    <button onClick={handleClick}>Go to About</button>
  );
}
```

Route Protection

To protect routes, you can use authentication or authorization mechanisms. For example, you could use a custom hook to check if the user is authenticated before rendering a protected route.

By understanding these concepts, you can effectively use React Router to manage navigation and create complex routing structures in your React applications.

2.2 Navigation using Links

Links in React Router are used to create clickable elements that, when clicked, navigate to a specific route within your application. This provides a seamless user experience for navigating between different pages or sections of your SPA.

Key Points:

- **Link component:** The Link component from react-router-dom is used to create links that navigate to specific routes.
- **to prop:** The to prop of the Link component specifies the path to navigate to. It can be a string representing the path or a path object with additional properties like search and hash.
- **NavLink component:** The NavLink component is a specialized version of Link that allows you to style active links differently. It accepts an activeClassName prop to specify the class to be applied to the active link.

Example:

```jsx
import { Link, NavLink } from 'react-router-dom';

function MyNavbar() {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <NavLink to="/about" activeClassName="active">About</Link>
        </li>
        <li>
          <NavLink to="/contact" activeClassName="active">Contact</Link>
        </li>
      </ul>
    </nav>
  );
}
```

In this example:

- The Link component is used to create links to the / and /about routes.

- The NavLink component is used to create links to the /about and /contact routes, with the activeClassName prop specifying the class to be applied to the active link.

Additional Considerations:

- **Nested Routes:** You can use nested routes and NavLink components to create complex navigation structures within your application.
- **Programmatic Navigation:** While Link components are primarily used for declarative navigation, you can also use programmatic navigation with useNavigate or the history object for more complex scenarios.
- **Styling:** You can customize the appearance of links using CSS or styling libraries like Styled Components or Emotion.

By effectively using Links in React Router, you can provide a seamless and intuitive navigation experience for your users.

2.2.1 404 page (Not found Page)

Creating a 404 Page in React Router

A 404 page is a helpful component that displays a message when a user tries to access a route that doesn't exist. This prevents users from seeing error messages and provides a more user-friendly experience.

Steps:

1. **Create a 404 Component:** Create a React component named NotFoundPage or similar. This component will render the content you want to display when a route is not found.
2. **Define the 404 Route:** In your main routing configuration, add a Route component with the path="*", which matches any URL that doesn't have a corresponding route defined. Set the element prop of this route to your NotFoundPage component.

Example:

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

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage    />} />
        <Route path="/about" element={<AboutPage />} />
        <Route path="/contact" element={<ContactPage    />} />
        <Route path="*" element={<NotFoundPage />} />
      </Routes>
    </Router>
  );
}
```

In this example, the NotFoundPage component will be rendered whenever a user tries to access a route that doesn't exist.

Customizing the 404 Page:

You can customize the appearance and content of your 404 page to match your application's design and provide helpful information to the user. Consider including the following:

- **Error message:** Clearly indicate that the requested page was not found.
- **Link to homepage or search:** Provide a link to the homepage or a search functionality to help users navigate back to relevant content.
- **Additional information:** You can optionally include additional information, such as a contact email or a link to frequently asked questions.

Example 404 Page:

```jsx
function NotFoundPage() {
  return (
    <div>
      <h1>404 - Page Not Found</h1>
      <p>The page you are looking for could not be found.</p>
      <Link to="/">Go to Homepage</Link>
    </div>
  );
}
```

By following these steps and customizing the NotFoundPage component, you can provide a more informative and user-friendly experience for users who encounter invalid routes in your React application.

2.2.2 URL Parameters

URL parameters are placeholders within a URL that can be replaced with dynamic values. They allow you to create more flexible and dynamic routes, making your application more adaptable to different scenarios.

Key Points:

- **Path Parameters:** Path parameters are enclosed in colons within the route path. For example, /users/:userId.
- **Accessing Parameters:** You can access the values of path parameters within your component using the useParams hook or the match prop.
- **Dynamic Routing:** URL parameters enable you to create dynamic routes that can handle different URL patterns, making your application more flexible.

Example:

```jsx
import { useParams } from 'react-router-dom';

function UserProfile() {
  const { userId } = useParams();

  // Fetch user data based on userId
  const userData = fetchUserData(userId);

  return (
    <div>
      <h1>User Profile</h1>
      <p>User ID: {userId}</p>
      {/* Render user data */}
    </div>
  );
}
```

In this example:

- The useParams hook is used to access the userId parameter from the URL.
- You can use the userId to fetch user data from an API or database.
- The fetched user data can then be rendered in the component.

Additional Considerations:

- **Validation:** It's important to validate URL parameters to ensure that they are valid and prevent unexpected behavior.
- **Security:** Be cautious when using URL parameters to avoid security vulnerabilities, such as injection attacks.
- **Dynamic Routes:** URL parameters can be combined with other routing features like nested routes to create complex routing structures.

By understanding and effectively using URL parameters in React Router, you can create more flexible and dynamic applications that can handle various scenarios and provide a better user experience.

2.2.3 Nested Routes

Nested Routes in React Router

Nested routes in React Router allow you to create hierarchical routing structures within your application. This enables you to organize your routes more effectively and provide a better user experience.

Key Points:

- **Nested Route Components:** To create nested routes, you can nest Route components within other Route components.
- **Parent-Child Relationship:** The nested Route components are considered children of their parent Route component.
- **Path Matching:** The path of a nested route is relative to its parent route.
- **Dynamic Routing:** You can use path parameters in nested routes to create dynamic routes.

Example:

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

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<HomePage />}>
          <Route path="about" element={<AboutPage />} />
          <Route path="contact" element={<ContactPage />} />
        </Route>
      </Routes>
    </Router>
  );
}
```

In this example:

- The HomePage component is the parent route.
- The AboutPage and ContactPage components are nested within the HomePage route.

- The paths of the nested routes are relative to the parent route. For example, the path /about refers to /about relative to the / path of the HomePage component.

Additional Considerations:

- **Dynamic Routes:** You can use path parameters in nested routes to create dynamic routes. For example, /products/:productId would be a nested route within the /products route.
- **Route Protection:** You can apply route protection to nested routes to restrict access to certain parts of your application.
- **Breadcrumbs:** Nested routes can be used to implement breadcrumbs navigation, which helps users understand their current location within the application.

By understanding and effectively using nested routes in React Router, you can create more organized and maintainable routing structures in your applications.

2.2.4 Implementing styles using NavLink

The NavLink component in React Router allows you to apply styles to active links, providing a visual indication to users about their current location within the navigation menu.

Key Points:

- **activeClassName Prop:** The activeClassName prop of the NavLink component specifies the CSS class to be applied to the link when it is active.
- **Custom Styles:** You can define custom CSS classes to style the active link in your application's stylesheet.
- **Dynamic Styling:** You can dynamically update the styles of the active link based on component state or other factors.

Example:

```jsx
import { NavLink } from 'react-router-dom';

function MyNavbar() {
  return (
    <nav>
      <ul>
        <li>
          <NavLink to="/" className="nav-link">Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" className="nav-link" activeClassName="active">About</NavLink>
        </li>
        <li>
          <NavLink to="/contact" className="nav-link" activeClassName="active">Contact</NavLink>
        </li>
      </ul>
    </nav>
  );
}
```

In this example:

- The NavLink components have the className prop set to nav-link, which defines the base styles for all links.
- The activeClassName prop is set to active, which will be applied to the active link.
- You can define the nav-link and active classes in your application's stylesheet to customize the appearance of the links.

Additional Considerations:

- **CSS Frameworks:** You can use CSS frameworks like Bootstrap or Material-UI to provide pre-defined styles for navigation links.
- **Styled Components:** If you're using a CSS-in-JS library like Styled Components, you can define styles directly within the NavLink component.
- **Dynamic Styling:** You can dynamically update the styles of the active link based on component state or other factors using JavaScript or state management libraries.

By effectively styling NavLink components, you can create visually appealing and informative navigation menus in your React Router applications.

2.3 Application Programming Interface

APIs are fundamental to modern web applications, enabling communication and data exchange between different components and systems. In React, APIs are used to:

- **Fetch data:** Retrieve data from external sources, such as databases, servers, or third-party services.
- **Send data:** Submit data to external systems, such as for form submissions, user authentication, or creating new resources.
- **Integrate with other services:** Connect your React application with other services, like payment gateways, social media platforms, or mapping APIs.

Common API Patterns:

- **RESTful APIs:** Represent resources as URLs and use HTTP methods (GET, POST, PUT, DELETE) to interact with them.
- **GraphQL APIs:** Provide a flexible way to fetch data by allowing clients to specify exactly what data they need.

Key Considerations:

- **API Design:** Choose an API design that is well-structured, easy to understand, and efficient.
- **Data Formatting:** Ensure that the data returned by APIs is in a format that your React application can easily consume (e.g., JSON).
- **Error Handling:** Implement proper error handling mechanisms to gracefully handle API errors and provide informative feedback to the user.
- **Caching:** Implement caching strategies to improve performance and reduce the load on your API.
- **Security:** Protect your API endpoints from security vulnerabilities like unauthorized access and injection attacks.

Popular Libraries for API Integration:

- **Axios:** A popular HTTP client library for making requests to RESTful APIs.

- **Apollo Client:** A GraphQL client library for interacting with GraphQL APIs.
- **Fetch API:** A built-in browser API for making network requests.

Example using Axios:

```
import axios from 'axios';

function MyComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get('https://api.example.com/data');
        setData(response.data);
      } catch    (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      {data && <pre>{JSON.stringify(data, null, 2)}</pre>}
    </div>
  );
}
```

In this example, the useEffect hook is used to fetch data from the specified API endpoint using Axios. The fetched data is then stored in the component's state and rendered to the UI.

By effectively using APIs in your React applications, you can leverage the power of external services and create more dynamic and interactive experiences for your users.

2.3.1 Build a REST API using json-server

json-server

json-server is a simple and lightweight tool that allows you to quickly create RESTful APIs based on JSON data files. It's a great option for prototyping or developing small-scale APIs without the need for a full-fledged database.

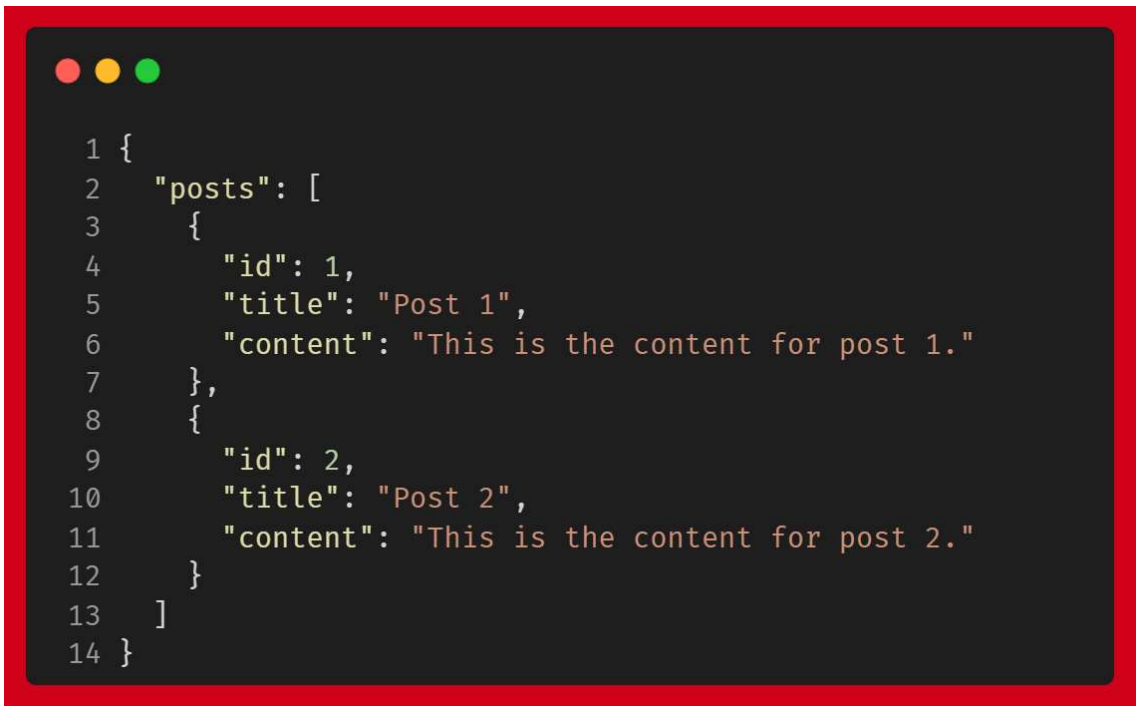
Installation:

$> npm install -g json-server

Creating a JSON Data File:

Create a JSON file (e.g., db.json) to store your data. For example:

JSON



```json
{
  "posts": [
    {
      "id": 1,
      "title": "Post 1",
      "content": "This is the content for post 1."
    },
    {
      "id": 2,
      "title": "Post 2",
      "content": "This is the content for post 2."
    }
  ]
}
```

Running json-server:

$> json-server --watch db.json --port 3000

This command will start a JSON server on port 3000, watching for changes to the db.json file and automatically updating the API.

Example API Endpoints:

- **GET /posts:** Retrieves a list of all posts.
- **GET /posts/:id:** Retrieves a specific post by its ID.
- **POST /posts:** Creates a new post.

- **PUT /posts/:id:** Updates an existing post.
- **DELETE /posts/:id:** Deletes an existing post.

Using the API in React:

```jsx
import axios from 'axios';

function MyComponent() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        const response = await axios.get('http://localhost:3000/posts');
        setPosts(response.data);
      } catch (error) {
        console.error('Error fetching posts:', error);
      }
    };

    fetchPosts();
  }, []);

  return (
    <div>
      {posts.map((post) => (
        <div key={post.id}>
          <h2>{post.title}</h2>
         <p>{post.content}</p>
        </div>
      ))}
    </div>
  );
}
```

In this example, the useEffect hook is used to fetch the list of posts from the JSON server and store them in the component's state. The fetched data is then rendered to the UI.

Key Considerations:

- **Data Structure:** Organize your data in a logical and consistent manner within the JSON file.
- **Error Handling:** Implement proper error handling to handle potential errors during API requests.
- **Security:** If you're using json-server in a production environment, consider security measures like authentication and authorization.

- **Scaling:** For larger applications or production environments, you may need to consider using a more robust database and API platform.

By following these steps and considering the key considerations, you can effectively use json-server to build RESTful APIs for your React applications.

2.3.2 API consumption in React application using Fetch method

Fetch API

The Fetch API is a built-in browser API that provides a way to make network requests and fetch data from external sources. It's a powerful tool for integrating your React applications with APIs.

Key Points:

- **Asynchronous Requests:** The Fetch API is asynchronous, meaning it doesn't block the main thread while waiting for the response.
- **Promises:** The fetch() method returns a Promise, which can be resolved or rejected depending on the success of the request.
- **Response Object:** The resolved Promise contains a Response object, which provides methods to access the response headers, status, and body.
- **JSON Data:** To parse JSON data from the response, use the json() method.

Example:

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

function MyComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async    () => {
      try {
        const response = await fetch('https://api.example.com/data');
        const json = await response.json();
        setData(json);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      {data && <pre>{JSON.stringify(data, null, 2)}</pre>}
    </div>
  );
}
```

In this example:

- The useEffect hook is used to fetch data from the specified API endpoint.
- The fetch() method is used to make the request.
- The json() method is used to parse the JSON response.
- The fetched data is stored in the component's state and rendered to the UI.

Additional Considerations:

- **Error Handling:** Implement proper error handling to catch and handle potential errors during API requests.
- **Caching:** Consider using caching strategies to improve performance and reduce the load on your API.
- **CORS:** If you're making requests to a different domain, ensure that the server allows Cross-Origin Resource Sharing (CORS).
- **HTTP Headers:** You can set custom HTTP headers using the headers option of the fetch() method.

By effectively using the Fetch API, you can integrate your React applications with various APIs and fetch data from external sources to create dynamic and interactive experiences.

[Practical Application of Unit-2: Build a dynamic Music Store application using Routing and API connectivity]

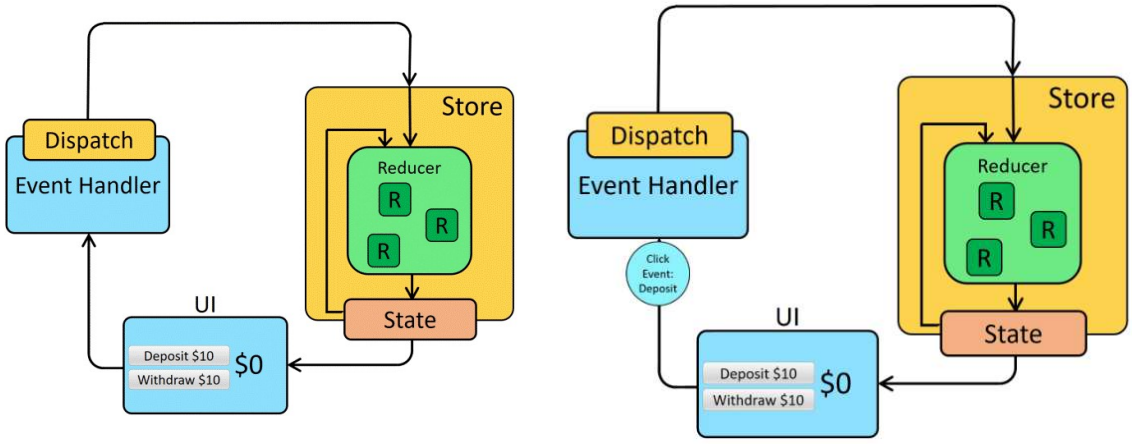
Unit 3. Redux and Saga-Middleware

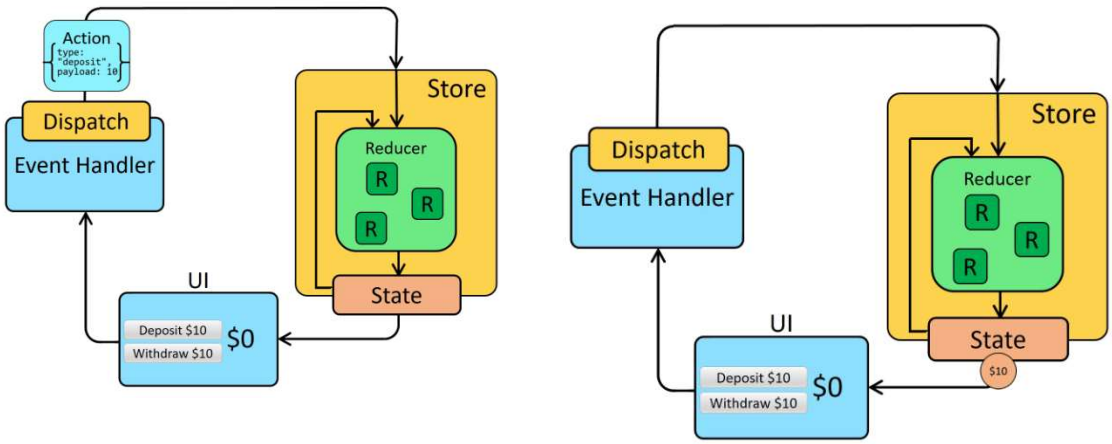
3.1 Redux: Need of Redux, Redux Architecture, Redux Action

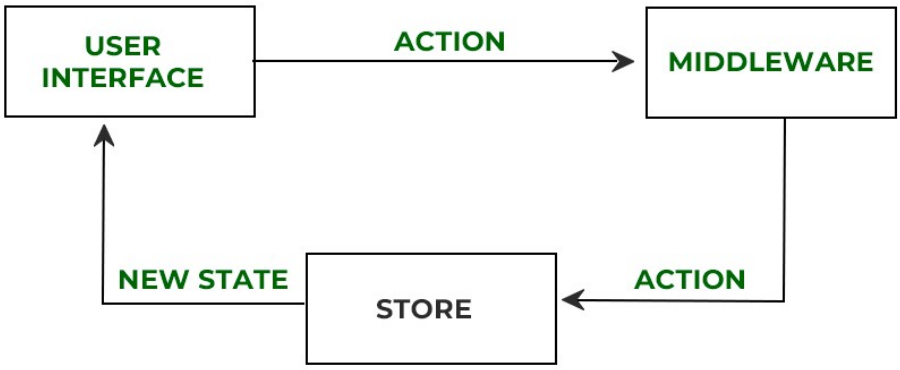
Need of Redux:

- **State Management:** In large applications, managing state across multiple components can become complex. Redux centralizes the state in a single store, making it easier to manage.
- **Predictability:** Redux ensures that the state transitions are predictable by enforcing strict rules on how changes can occur.
- **Debugging:** Redux's centralized state makes it easier to track and debug changes.
- **Testability:** Redux's architecture simplifies unit testing by isolating the logic from the components.

Redux Architecture:







- **Store:** A single JavaScript object that holds the application's state.
- **Actions:** Plain JavaScript objects that describe what happened in the application.
- **Reducers:** Pure functions that take the current state and an action, and return a new state.
- **Dispatch:** The process of sending an action to the store.
- **Middleware:** Functions that allow actions to be intercepted and modified before they reach the reducers.

Redux Action:

- **Definition:** An action is an object that contains a type field and may include additional data. The type field describes the type of action, and the data provides information needed to perform the action.

Example:

```
const incrementAction = {
  type: 'INCREMENT',
  payload: 1
};
```

3.1.1 Redux Reducers, Redux Store, Principles of Redux

Redux Reducers:

- **Function:** Reducers are pure functions that take the current state and an action, then return a new state based on the action type.

Example:

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

Redux Store:

- **Creation:** The store is created using the createStore function from Redux.

- **Responsibilities:**
    - Holds the application state.
    - Allows access to the state via getState().
    - Allows state to be updated via dispatch(action).
    - Registers listeners via subscribe(listener).
    - Handles the unregistration of listeners via the function returned by subscribe(listener).

Principles of Redux:

1. **Single Source of Truth:** The entire state of the application is stored in a single object tree within a single store.
2. **State is Read-Only:** The only way to change the state is to emit an action, an object describing what happened.
3. **Changes are Made with Pure Functions:** To specify how the state tree is transformed by actions, you write pure reducers.

3.1.2 Pros of Redux

Pros of Redux:

- **Centralized State Management:** Makes state predictable and consistent across the application.
- **Easier Debugging:** With tools like Redux DevTools, you can track state changes in real-time.
- **Improved Testability:** Since reducers are pure functions, they are easier to test.
- **Flexible Middleware:** Middleware like Redux Thunk or Saga allows for handling asynchronous operations and other side effects.

3.2 NPM Packages required to work with Redux

When setting up Redux in a React application, there are several key NPM packages that you will typically need:

3.2.1 redux

- **Purpose:** The core Redux library, which provides the foundational tools to create a Redux store, define reducers, and manage the application state.
- **Installation:**

$> npm install redux

react-redux

- **Purpose:** The official React binding for Redux, which provides components and hooks to connect your React components to the Redux store.
- **Key Features:**
    - Provider: A component that wraps your React app and makes the Redux store available to all components.
    - useSelector: A hook to extract data from the Redux store state.
    - useDispatch: A hook to dispatch actions to the Redux store.
- **Installation:**

npm install react-redux

redux-thunk (optional)

- **Purpose:** A middleware that allows you to write action creators that return a function instead of an action. This is useful for handling asynchronous logic in Redux, such as API calls.
- **Installation:**
- npm install redux-thunk
- **Usage:**
    - In your Redux store setup, apply redux-thunk as middleware:

```
1 import { createStore, applyMiddleware } from 'redux';
2 import thunk from 'redux-thunk';
3 import rootReducer from './reducers';
4 const store = createStore(rootReducer, applyMiddleware(thunk));
```

redux-saga (optional)

- **Purpose:** An alternative to redux-thunk, Redux Saga is a middleware that helps to handle side effects (like asynchronous actions) using generator functions, making it easier to manage complex asynchronous workflows.
- **Installation:**

npm install redux-saga

- **Usage:**
  - In your Redux store setup, apply redux-saga as middleware:

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';
import rootReducer from './reducers';
import rootSaga from './sagas';

const sagaMiddleware = createSagaMiddleware();
const store = createStore(rootReducer, applyMiddleware(sagaMiddleware));
sagaMiddleware.run(rootSaga);
```

@reduxjs/toolkit (optional but recommended)

- **Purpose:** A toolkit provided by the Redux team to simplify the setup and use of Redux in an application. It includes utilities like createSlice and configureStore that reduce boilerplate code.
- **Key Features:**
  - createSlice: Automatically generates actions and reducers.
  - configureStore: Sets up the store with good defaults, including Redux DevTools integration.
  - Built-in support for redux-thunk.
- **Installation:**

  $> npm install @reduxjs/toolkit

These packages will provide everything needed to set up and work with Redux in your React application. Would you like to explore any specific examples or details related to these packages?

3.3 Async Operations:

3.3.1 Need of Async operations

- Redux primarily focuses on managing state changes in a synchronous manner. However, many real-world applications involve asynchronous operations like fetching data from APIs, making network requests, or performing time-consuming calculations.
- These asynchronous operations cannot be directly handled within reducers, as reducers must be pure functions that return a new state without modifying the existing one.
- To address this, Redux provides mechanisms to handle asynchronous operations effectively.

3.3.2 Async Workflow

The typical async workflow in Redux involves the following steps:

1. **Dispatch an Action:** An action is dispatched to initiate the asynchronous operation. This action can be a plain JavaScript object or a function that returns a promise.
2. **Middleware Interception:** Middleware like redux-thunk or redux-saga intercepts the action and performs the asynchronous operation.
3. **Promise Resolution:** The asynchronous operation resolves with a result or an error.
4. **Dispatch Success/Failure Actions:** Based on the result, the middleware dispatches success or failure actions with appropriate payload.
5. **Reducer Update:** The reducers handle the success or failure actions, updating the state accordingly.

3.3.3 Action Creators and How to write Action Creators

- Action creators are functions that return action objects. They are often used to encapsulate the logic for creating actions, making it easier to reuse and maintain.

- Action creators can be synchronous or asynchronous. Synchronous action creators return plain JavaScript objects, while asynchronous action creators typically return promises.

Example of a synchronous action creator:

```
function fetchUserData(userId) {
  return {
    type: 'FETCH_USER_DATA',
    payload: userId,
  };
}

```

Example of an asynchronous action creator using redux-thunk:

```
import axios from 'axios';

function fetchUserData(userId) {
  return (dispatch) => {
    dispatch({ type: 'FETCH_USER_DATA_REQUEST' });

    return axios
      .get(`/users/${userId}`)
      .then((response) => {
        dispatch({ type: 'FETCH_USER_DATA_SUCCESS', payload: response.data });
      })
      .catch((error) => {
        dispatch({ type: 'FETCH_USER_DATA_FAILURE', payload: error    });
      });
  };
}
```

3.3.4 Handling Async Actions via Reducers

- Reducers handle the success and failure actions dispatched by the middleware.
- They update the state based on the payload of these actions, reflecting the results of the asynchronous operation.
- Reducers should typically handle the initial request, success, and failure states to provide a clear indication of the asynchronous operation's progress.

Example of a reducer handling async actions:

```
const initialState = {
  userData: null,
  loading: false,
  error: null,
};

function userReducer(state = initialState, action) {
  switch (action.type) {
    case 'FETCH_USER_DATA_REQUEST':
      return { ...state, loading: true };
    case 'FETCH_USER_DATA_SUCCESS':
      return { ...state,    loading: false, userData:    action.payload };
    case 'FETCH_USER_DATA_FAILURE':
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
}
```

By following these guidelines, you can effectively handle asynchronous operations in Redux, ensuring a seamless and predictable user experience.

3.4 Middleware: Redux-Saga

3.4.1 Generators in Redux-Saga

- **What are Generators?**
    - Generators are a special type of function in JavaScript that can pause and resume execution. They are defined using the function* syntax and can yield multiple values over time.

Example:

```
function* myGenerator() {
  yield 'First value';
  yield 'Second value';
  return 'Final value';
}

const gen = myGenerator();
console.log(gen.next().value); // 'First value'
console.log(gen.next().value); // 'Second value'
console.log(gen.next().value); // 'Final value'
```

- **Use in Redux-Saga:**
    - Redux-Saga uses generators to handle side effects in a declarative way. Instead of executing side effects like API calls directly, a saga yields effects, which Redux-Saga interprets and executes.

3.4.2 Saga Methods()

- **Common Saga Methods:**
    - **takeEvery**: Watches for dispatched actions and runs the provided saga for each action.

```
import { takeEvery } from 'redux-saga/effects';
function* fetchDataSaga() {
  // code to fetch data
}

function* watchFetchData() {
  yield takeEvery('FETCH_DATA_REQUEST', fetchDataSaga);
}
```

    - **takeLatest**: Similar to takeEvery, but only runs the latest saga if multiple actions are dispatched in quick succession.

```
import { takeLatest } from 'redux-saga/effects';

function* fetchDataSaga() {
  // code to fetch data
}

function* watchFetchData() {
  yield takeLatest('FETCH_DATA_REQUEST', fetchDataSaga);
}
```

    - **call**: Calls a function and waits for it to resolve, useful for executing asynchronous operations like API requests.

```
import { call } from 'redux-saga/effects';

function fetchDataApi() {
  return fetch('/api/data').then(response => response.json());
}

function* fetchDataSaga() {
  const data = yield call(fetchDataApi);
  // code to handle data
}
```

- **put**: Dispatches an action to the Redux store, similar to dispatch in Redux.

```
import { put } from 'redux-saga/effects';

function* fetchDataSaga() {
  try {
    const data = yield call(fetchDataApi);
    yield put({ type: 'FETCH_DATA_SUCCESS', payload: data });
  } catch (error) {
    yield put({ type: 'FETCH_DATA_FAILURE', payload: error.message });
  }
}
```

- **all**: Runs multiple sagas concurrently.

```
import { all } from 'redux-saga/effects';

function* rootSaga() {
  yield all([
    watchFetchData(),
    // other watchers
  ]);
}
```

3.4.3 Building a Product List

- **Setup:**
  - **Action Types:**

```
const FETCH_PRODUCTS_REQUEST = 'FETCH_PRODUCTS_REQUEST';
const FETCH_PRODUCTS_SUCCESS = 'FETCH_PRODUCTS_SUCCESS';
const FETCH_PRODUCTS_FAILURE = 'FETCH_PRODUCTS_FAILURE';
```

  - **Action Creators:**

```
const fetchProductsRequest = () => ({ type: FETCH_PRODUCTS_REQUEST });
const fetchProductsSuccess = (products) => ({ type: FETCH_PRODUCTS_SUCCESS,
payload: products });
const fetchProductsFailure = (error) => ({ type: FETCH_PRODUCTS_FAILURE,
payload: error });
```

  - **Reducer:**

```
const initialState = {
  loading: false,
  products: [],
  error: ''
};

const productReducer = (state = initialState, action) => {
  switch (action.type) {
    case FETCH_PRODUCTS_REQUEST:
      return { ...state, loading: true };
    case FETCH_PRODUCTS_SUCCESS:
      return { ...state, loading: false, products: action.payload };
    case FETCH_PRODUCTS_FAILURE:
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
};
```

- **Saga:**

```
import { call, put, takeEvery } from 'redux-saga/effects';

function fetchProductsApi() {
  return fetch('/api/products').then(response => response.json());
}

function* fetchProductsSaga() {
  try {
    const products = yield call(fetchProductsApi);
    yield put(fetchProductsSuccess(products));
  } catch (error) {
    yield put(fetchProductsFailure(error.message));
  }
}

function* watchFetchProducts() {
  yield takeEvery(FETCH_PRODUCTS_REQUEST, fetchProductsSaga);
}
```

- **Root Saga:**

```
import { all } from 'redux-saga/effects';

function* rootSaga() {
  yield all([
    watchFetchProducts(),
    // other sagas
  ]);
}
```

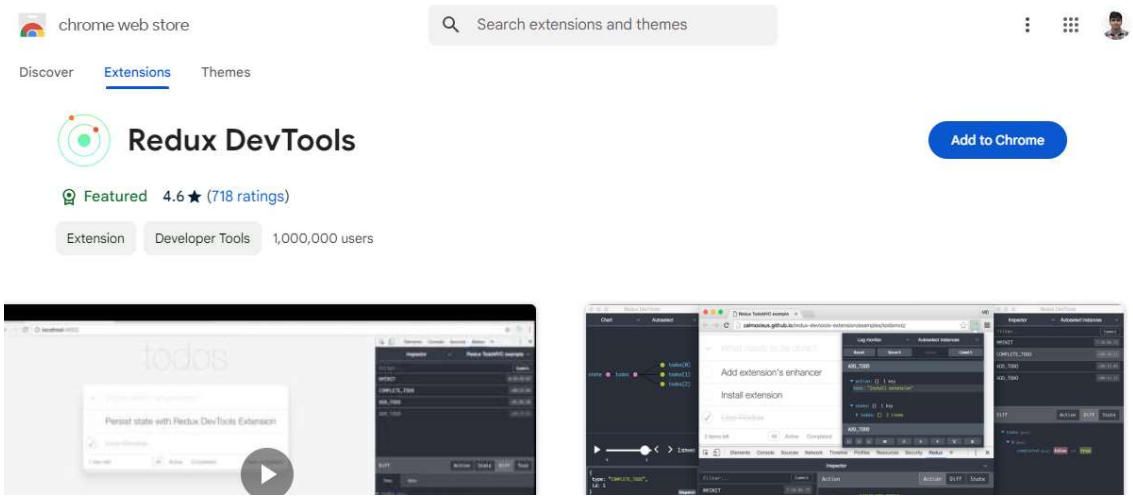
- **Store Configuration:**

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga';
import productReducer from './reducers';
import rootSaga from './sagas';

const sagaMiddleware = createSagaMiddleware();
const store = createStore(productReducer,
applyMiddleware(sagaMiddleware));

sagaMiddleware.run(rootSaga);
```

- **Usage in React Component:**

```jsx
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchProductsRequest } from './actions';

const ProductList = () => {
  const dispatch = useDispatch();
  const { loading, products, error } = useSelector(state => state);

  useEffect(() => {
    dispatch(fetchProductsRequest());
  }, [dispatch]);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <ul>
      {products.map(product => (
        <li key={product.id}>{product.name}</li>
      ))}
    </ul>
  );
};

export default ProductList;
```

3.4.4 Debugging application using Redux Devtools



- **Integration:**
  - Redux DevTools is a powerful tool for debugging Redux applications. It allows you to inspect every action, view the state at any point in time, and even time-travel through the state changes.
  - **Installation:**

    $> npm install redux-devtools-extension
  - **Setup:**

```
1 import { createStore, applyMiddleware } from 'redux';
2 import { composeWithDevTools } from 'redux-devtools-extension';
3 import createSagaMiddleware from 'redux-saga';
4 import productReducer from './reducers';
5 import rootSaga from './sagas';
6
7 const sagaMiddleware = createSagaMiddleware();
8 const store = createStore(productReducer,
  composeWithDevTools(applyMiddleware(sagaMiddleware)));
9
10 sagaMiddleware.run(rootSaga);
```

- **Features:**
  - **State Inspection:** View the current state of your Redux store.
  - **Action Tracking:** See a list of all actions dispatched in your application, along with the payload and the resulting state.

- **Time Travel:** Navigate through past states and replay actions to understand the flow of your application.

[Practical Application of Unit-3: Building an application to list the food items using React and Redux. Building News application using React, Redux, and promise middleware. Building a Product list application using Redux-Saga Middleware.]