Understanding React JS

By Arun Vijayarengan, Founder, Edsteem

Understanding React JS

By Arun Vijayarengan, Founder, Edsteem

Welcome to the exciting world of React JS! Whether you're a seasoned developer or just starting your coding adventure, React offers a powerful and efficient way to create dynamic and interactive web applications.

Table of Contents

- Quick Introduction to React JS
- Setting Up Your Dev Machine for React JS App Development
- File Walkthrough of the React Project
- Chapter 1: Components
- Chapter 2: Thinking in React
- Chapter 3: JSX
- Chapter 4: Props in React
- Chapter 5: States with React Hooks
- Chapter 6: Events in React
- Chapter 7: Conditional Rendering in React
- Chapter 8: Lists and Keys in React
- Chapter 9: Routing with react-router-dom
- Chapter 10: Forms in React
- Chapter 11: The useEffect Hook and API Calls
- Chapter 12: Higher-Order Components with Functional Components
- Chapter 13: Fragments in React
- Chapter 14: Error Boundaries in React
- Chapter 15: Sharing Data Between Components In React
- Chapter 16: Hooks, Rules of Hooks, Advanced Hooks and Custom Hook
- Chapter 17: Building more complex navigation and routing structures
- Chapter 18: Authentication in React JS
- Chapter 19: Code splitting
- Chapter 20: Internationalization and Localization

- Chapter 21: Build and Deployment
- Chapter 22: Performance Optimisation Tips in React JS App
- Bonus Chapter 23: Styling in React with Styled Components

Quick Introduction to React JS:

React JS, developed by Facebook, is a JavaScript library designed for building efficient and reusable user interfaces. Its core philosophy centers around the concept of components, enabling developers to break down complex UIs into modular and manageable pieces. React's declarative approach to programming allows developers to describe how the UI should look in different states, and React takes care of efficiently updating and rendering the components as the application state changes. With its Virtual DOM, React ensures optimal performance by minimizing unnecessary re-renders and efficiently updating only the parts of the DOM that need modification.

Component-Based Architecture:

At the heart of React is its component-based architecture, promoting the creation of encapsulated and reusable UI elements. Components can be composed and nested, making it easy to build and maintain large-scale applications. The reusability of components not only streamlines development but also enhances collaboration as different teams can work on isolated components concurrently.

State Management and Reactivity:

React's state management, facilitated by concepts like props and state, provides a robust mechanism for handling data changes. The unidirectional data flow ensures

a predictable and straightforward approach to managing application state. Reactivity is a key aspect, allowing UI components to automatically update in response to changes in data, providing a seamless and dynamic user experience.

Thriving Ecosystem and Community:

React has cultivated a vibrant ecosystem with a plethora of libraries and tools. The React ecosystem includes state management libraries like Redux, routing with React Router, and styling solutions like Styled Components. The extensive community support ensures a wealth of resources, tutorials, and a constant influx of innovations, contributing to React's continual evolution and adaptation to industry needs.

What are the Competing Tools of React JS?

1. Vue.js

Vue.js, a progressive JavaScript framework, is renowned for its simplicity and ease of integration. With a reactive data-binding system and a component-based architecture similar to React, Vue appeals to developers seeking a lightweight alternative for building modern user interfaces. Its gentle learning curve and flexible design make it a compelling choice for small to medium-sized projects.

2. Angular

Angular, developed and maintained by Google, is a comprehensive and opinionated framework for building robust web applications. Unlike React and Vue, Angular employs a more prescriptive approach to development, offering a complete solution with features like two-way data binding, dependency injection, and a powerful CLI. Angular is favored for large-scale enterprise applications where a strict architectural framework is crucial.

3. Svelte:

Svelte takes a different approach by shifting the work from the client-side to build time. With Svelte, components are compiled into highly optimized JavaScript at build, resulting in smaller bundle sizes and improved runtime performance. Svelte's simplicity and focus on efficiency make it an intriguing choice for developers looking to streamline the development process and optimize for performance.

Setting Up Your Dev Machine for React JS App Development

Developing React applications efficiently starts with a well-configured development environment. Follow these steps to set up your machine for React JS development.

1. Install Node.js and npm

React relies on Node.js and npm (Node Package Manager) for managing dependencies and running scripts. Visit the official Node.js website to download and install the latest LTS version. npm comes bundled with Node.js, and together they form the foundation for building and running React applications.

After installation, verify the versions by running `node -v` and `npm -v` in your terminal / command prompt.

2. Choose a Code Editor

Selecting a code editor suited for React development is crucial. Popular choices include Visual Studio Code, Atom, and Sublime Text. Visual Studio Code, in particular, offers excellent support for React with extensions like "ESLint" for code linting and "Prettier" for code formatting. Customize your editor to enhance your workflow, and leverage features like integrated terminals, Git integration, and debugging tools.

3. Create a React App with Create React App

The easiest way to kickstart a React project is by using Create React App, a tool that sets up a new React project with sensible defaults and a pre-configured build system.

Create a new React app with `npx create-react-app my-react-app` (replace "my-react-app" with your project's name).

This command installs the necessary dependencies and generates a project structure for you.

4. Additional Tools for React Development

Enhance your development experience by integrating additional tools. The React DevTools browser extension is invaluable for inspecting and debugging React components. Install it for Chrome or Firefox to gain insights into component hierarchies, states, and props. Consider using a state management library like Redux for more complex state requirements. Extensions like React-Router will aid in creating navigable single-page applications. Lastly, explore testing frameworks like Jest and React Testing Library to ensure the reliability of your code.

With a well-configured development machine, you're ready to dive into React app development. Regularly update your tools and stay informed about the latest advancements in the React ecosystem to optimize your workflow and stay ahead in the ever-evolving field of web development.

File Walkthrough of the React Project

When you create a new React project using the create-react-app tool, it generates a standardized project structure with various files and directories. Here's a walkthrough of the key files and directories commonly found in a React project created with create-react-app:

node*modules/*

This directory contains all the project dependencies installed using npm. You don't need to manually manage this directory; npm handles it for you.

public/

The public directory holds static assets that you want to include in the build process but don't need to be processed by Webpack. The index.html file in this directory is the entry point for your React application.

src/

The src directory is where you'll spend most of your time writing code. It contains the source code for your React application.

src/index.js:

The entry point for your React application. It renders the App component into the DOM.

src/index.css:

Global styles that apply to the entire application. You can modify or extend this file to define global styles.

src/App.js:

The main component of your application. It can be modified or replaced with your own components to structure your UI.

src/App.css:

The stylesheet for the App component, containing styles specific to this component.

src/logo.svg:

An example SVG file used in the default App component. You can replace it with your own assets.

src/serviceWorker.js:

A service worker script for caching assets and providing offline capabilities. It's optional and can be configured based on your project requirements.

src/setupTests.js:

Configuration for Jest, the testing library used by create-react-app.

package.json and package-lock.json:

These files define your project's dependencies, scripts, and metadata. package-lock.json ensures that dependencies are installed with consistent versions across different machines.

public/index.html:

The HTML file that serves as the main entry point for your React application. It includes the root div where React will render your components.

public/manifest.json:

A JSON file that provides metadata about the web application. It includes information such as the app's name, icons, and other properties for progressive web apps.

README.md:

The README file provides information about your project, including instructions on how to run and build the application.

This is a basic overview of the most important files and directories in a React project created with create-react-app. As you build your application, you may create additional files and directories based on your project's specific needs.

Chapter 1: Components

React JS is a powerful JavaScript library used for building user interfaces. It's known for its flexibility, performance, and component-based architecture. In this chapter, we'll dive into the basics of React, starting with Components.

In React, a component is a fundamental building block that encapsulates a part of a user interface and its behaviour. Components are reusable, self-contained, and can be composed to create complex user interfaces. There are two main types of components in React:

1. Functional Components

Functional Components are the simplest way to define a component in React. They are essentially JavaScript functions that return React elements. They are recommended in the last few years.

2. Class Components

Class Components are defined as ES6 classes. **Class Components have been widely used in earlier versions of React**, though with the introduction of React Hooks in React JS Version 16.8, functional components have gained more popularity and have largely replaced class components for new development. Official React JS documentation doesn't recommend Class Components any more from React JS Version 18. So, we will not learn about them in this Book.

Functional Components

Let's create a basic functional component to understand this concept:

```
import React from 'react';
```

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

In the example above, we've defined a Welcome functional component that takes props as its input and returns a JSX element.

Chapter 2: Thinking in React

When building applications with React, it's important to think in terms of components. This approach helps in creating a modular and maintainable codebase. Here's a simplified approach to thinking in React:

Thinking in React is an important approach when building applications with React. It's about understanding how to structure your application into components, manage data flow, and create a modular and maintainable codebase. This concept is known as component-driven architecture. All you have to do is identify the different parts of your UI that can be broken down into reusable components.

Component-Driven Architecture

React encourages a component-driven approach to building user interfaces. This means breaking your UI into small, reusable components, each responsible for a specific part of the UI. Here's a step-by-step guide on how to think in React:

1. **Identify the UI Components**

Begin by identifying the different parts of your user interface that can be represented as components. For example, a web page may have a header, sidebar, content area, and a footer, each of which can be a component.

2. **Hierarchy and Nesting**

Organize these components hierarchically. Components can be parent or child components, and they should nest together logically. For instance, a web page

may have a Page component as the top-level parent, containing Header, Sidebar, Content, and Footer child components.

3. Data Flow

Determine how data flows within your components. Think about which components need certain data and how it should be passed from parent to child components. Props play a crucial role in this data flow.

4. State Management

Consider whether a component needs to manage its internal state. State can be controlled using React Hooks (e.g., useState).

5. Reusability

Strive for reusability. Components should be designed in a way that they can be reused in different parts of your application. This reduces code duplication and enhances maintainability.

Example: Creating a Product List

Let's take an example of a product list:

Identify Components: In this case, you might have components like ProductList, Product, and ProductDetail.

Hierarchy: ProductList might be the parent component that renders a list of Product components, which can be further nested to show the ProductDetail.

Data Flow: The ProductList component might fetch data from an API and pass it as props to Product components. Clicking on a product can trigger a state change in the ProductDetail component.

State Management: ProductDetail might manage the state of the selected product.

Reusability: Ensure that these components are designed in a way that they can be reused throughout your application.

Modular Development

Thinking in React also involves breaking down complex features into manageable pieces. Each piece is a component, and together they form the feature. This modular approach simplifies development, testing, and maintenance.

Thinking in React is a vital skill that empowers you to design modular, maintainable, and efficient applications. It involves breaking down your UI into components, considering data flow, and ensuring reusability. With practice, you'll become more adept at designing React applications with a clear and structured mindset.

In the next chapter, we'll explore the concept of props in greater detail, diving into how to pass and use data between React components.

Stay tuned for more React insights and hands-on examples!

Chapter 3: JSX

JSX is a fundamental part of React that allows developers to define the structure and appearance of UI elements. In this chapter, we will take an in-depth look at JSX, its syntax, and how it plays a vital role in building React applications.

What is JSX?

JSX is an extension of JavaScript that resembles HTML and allows you to describe the structure of your user interfaces in a declarative manner. It is not a separate language, but rather a syntax extension used by React to create React elements. Consider this example:

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

In the above code, element is a React element created using JSX, which can be rendered within your React components.

JSX Elements and Components

JSX allows you to create elements that represent UI components. React components can be written using a combination of JSX elements, which makes your code more readable and maintainable. For instance:

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

In the Welcome component, the JSX element <h1>Hello</h1> represents the user interface structure.

Embedding Expressions

JSX allows you to embed JavaScript expressions within curly braces {}. This makes it easy to include dynamic data within your UI. Here's an example:

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

const App = () => {
  const user = "John";
  return <Greet name={user} />;
};
```

In this case, props.name and user are JavaScript expressions embedded within the JSX code.

JSX Attributes

Just like HTML attributes, JSX allows you to add attributes to elements. These attributes can be used to pass data or configuration to a component. For example:

```
const element = <img src="my-image.jpg" alt="My Image" />;
```

Here, src and alt are attributes for the img element.

Self-Closing Tags

JSX supports self-closing tags, similar to HTML. For instance, an hr element in HTML can be represented as follows:

```
<hr />;
```

Conclusion

JSX is an integral part of React, enabling developers to create dynamic and expressive user interfaces. In this chapter, we've covered the basics of JSX, including its syntax, embedding expressions, using attributes, and representing React components.

As you go deeper into React, a solid understanding of JSX will be essential for creating engaging and interactive user interfaces. In the next chapter, we will explore the concept of props in greater detail and how they enable data flow within React components.

Chapter 4: Props in React

Props are a fundamental concept in React. They allow you to pass data from parent components to child components, making your React applications dynamic and re-usable. In this chapter, we'll go into the concept of props, how to use them, and their significance in building React applications.

What are Props?

Props are a way to pass data from a parent component to a child component. They are used to customize and configure child components, enabling them to display dynamic content and behavior. Think of props as the parameters that you pass to a function when creating an instance of a component.

Passing Props

To pass props to a child component, you specify them as attributes when rendering the child component. For example:

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

const App = () => {
  return <Greet name="John" />;
};
```

In this example, App component is parent component that hosts a child component 'Greet'. The 'name' prop is passed to the Child component Greet from Parent Component.

Accessing Props

Inside the child component, you can access props as properties of the props object. For example:

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

The `props.name` expression is used to access the name prop's value.

Dynamic Props

Props are not limited to static values. You can pass dynamic values, including variables and expressions, as props. For instance:

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

Here, the name prop is set to the value of the user variable.

Default Props

You can define default values for props in case they are not provided. This can be done using the defaultProps property:

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

Greet.defaultProps = {
  name: "Guest",
};
```

If the name prop is not provided when rendering Greet, it will default to "Guest."

PropTypes

To ensure the correct data type for props and enhance code robustness, you can use PropTypes. The prop-types library is often used to specify the expected prop types for a component. For example:

```
import PropTypes from 'prop-types'; // npm i prop-types

function Greet(props) {
  return <p>Hello, {props.name}!</p>;
}

Greet.propTypes = {
  name: PropTypes.string,
};
```

Props are a core concept in React, enabling the passing of data from parent to child components. They make components reusable and dynamic, allowing you to create flexible and configurable user interfaces.

In the next chapters, we'll explore states, conditional rendering, and lists with keys in React. These concepts are essential for building interactive user interfaces.

Chapter 5: States with React Hooks

In React, state represents the data that a component needs to keep track of. React Hooks, introduced in React 16.8, provide a way to manage state in functional components. In this chapter, we'll explore how to work with states using React Hooks.

Understanding Component State

State in a React component is a JavaScript object that stores data relevant to that component. It allows components to be dynamic and interactive. Let's create a simple counter component that demonstrates the use of state:

```
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>
  );
}
```

In this example, we use the useState hook to declare and initialize the count state variable. The setCount function is used to update the state, causing the component to re-render when the state changes.

The useState Hook

The useState hook is a fundamental hook in React for managing state in functional components. It takes an initial value as an argument and returns an array containing the current state value and a function to update it. Here's a breakdown of the useState hook:

```
const [state, setState] = useState(initialValue);
```

state: Represents the current state value.

setState: A function to update the state.

Managing Multiple States

You can use the useState hook multiple times in a component to manage different state variables. For example:

```
const [name, setName] = useState('John');
const [age, setAge] = useState(30);
```

Each state variable is independent and can be updated separately.

The useEffect Hook

The useEffect hook is another essential hook for managing side effects in functional components. It allows you to perform actions after the component renders, such as data fetching or subscriptions. Here's a basic usage of useEffect:

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

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

```
  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]);

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

In this example, we use useEffect to update the document title whenever the count state changes.

React Hooks provide a powerful way to manage state and side effects in functional components. We've explored the useState and useEffect hooks in this chapter, but there are more hooks available in React for specific use cases.

As you continue your React journey, understanding how to manage state is crucial for building dynamic and interactive applications. In the next chapter, we will cover events, conditional rendering, and lists with keys in React.

Chapter 6: Events in React

Events play a crucial role in making your React applications interactive and responsive. React provides a simple and efficient way to handle events, allowing you to respond to user actions and trigger specific behavior. In this chapter, we'll explore how to work with events in React.

Event Handling

React components can respond to various user interactions, such as clicks, keystrokes, and mouse movements. Event handling in React is similar to traditional JavaScript, but it is a bit more declarative and follows a specific pattern.

Handling Click Events

Let's start with a common event: the click event. To handle a click event, you define an event handler function and attach it to a React element using the onClick attribute. For example:

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

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

  const handleClick = () => {
    setCount(count + 1);
  };
```

```
  return (
    <div>
      <p>Click Count: {count}</p>
      <button onClick={handleClick}>Click me</button>
    </div>
  );
}
```

In this example, when the button is clicked, the handleClick function is called, updating the count state and re-rendering the component.

Handling Form Input

Handling form input is another common scenario in web applications. You can use the onChange event to capture and respond to changes in input fields. For instance:

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

function InputField() {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <div>
      <input
        type="text"
        value={inputValue}
        onChange={handleChange}
      />
      <p>You typed: {inputValue}</p>
    </div>
  );
```

```
}
```

In this example, the handleChange function is triggered when the input field's value changes, updating the inputValue state.

Preventing Default Behavior

React allows you to prevent the default behavior of certain events, such as form submissions or anchor clicks. To do this, you can use the preventDefault method. For example:

```
import React from 'react';

function PreventDefaultExample() {
  const handleClick = (event) => {
    event.preventDefault();
    alert('Link clicked, but default behavior prevented.');
  };

  return (
    <a href="https://www.example.com" onClick={handleClick}>
      Click me
    </a>
  );
}
```

In this example, the handleClick function prevents the default behavior (navigating to the linked URL) when the anchor is clicked.

React's declarative approach to event handling makes it straightforward to respond to user actions. In this chapter, we've explored how to handle click events, input changes, and how to prevent default behavior in React applications.

In the next chapter, we'll dive into conditional rendering and the use of lists with keys.

Chapter 7: Conditional Rendering in React

Conditional rendering is a fundamental concept in React that allows you to actions such as show or hide elements based on certain conditions. It's a key technique for building dynamic user interfaces. In this chapter, we'll dive into how conditional rendering works in React.

Using Conditional Statements

In React, you can use conditional statements like if and else to conditionally render components. For example, to render different content based on a condition, you can use the following pattern:

```
function ConditionalRender(props) {
  if (props.condition) {
    return <p>This content is rendered conditionally.</p>;
  } else {
    return <p>This content is an alternative.</p>;
  }
}
```

In this example, the content is rendered based on the value of the condition prop.

Using Ternary Operator

A more concise way to achieve conditional rendering is by using the ternary operator. This approach is especially useful for rendering one of two components or expressions based on a condition:

```
function ConditionalRender(props) {
  return (
    <div>
      {props.condition ? (
        <p>Render this content when true.</p>
      ) : (
        <p>Render this content when false.</p>
      )}
    </div>
  );
}
```

The ternary operator makes the code more readable and concise.

Conditional Rendering with Logical && Operator

Another approach for conditional rendering is to use the logical && operator. You can conditionally render elements when a condition is met:

```
function ConditionalRender(props) {
  return (
    <div>
      {props.condition && <p>Render this content when true.</p>}
    </div>
  );
}
```

In this example, the paragraph is rendered only when props.condition is true.

Conditional Rendering with Short-Circuit Evaluation

You can leverage short-circuit evaluation to conditionally render elements based on conditions:

```
function ConditionalRender(props) {
  return (
    <div>
      {props.condition && (
        <>
          <p>Render this paragraph when true.</p>
          <p>Render another paragraph when true.</p>
        </>
      )}
    </div>
  );
}
```

Here, multiple elements are conditionally rendered in one go when the condition is met.

Conditional rendering is a powerful technique that allows you to tailor your UI based on specific conditions. In React, you can use if statements, the ternary operator, the && operator, or short-circuit evaluation to achieve conditional rendering. This flexibility empowers you to create dynamic user interfaces.

In the next chapter, we will explore the concept of working with lists in React, including using keys for efficient rendering.

Chapter 8: Lists and Keys in React

Working with lists of data is a common task in React applications. React provides a straightforward way to render dynamic lists and optimize the rendering process using keys. In this chapter, we'll delve into how to work with lists and keys in React.

Rendering Lists

You can render a list of items in React by mapping through an array of data and generating a component for each item. For example, rendering a list of names:

```
function NameList(props) {
  const names = props.names;
  const nameListItems = names.map((name, index) => (
    <li key={index}>{name}</li>
  ));

  return <ul>{nameListItems}</ul>;
}
```

In this example, we use the map function to iterate through the names array and generate a list of <li> elements.

Using Keys

Keys are essential for React to efficiently update and re-render lists. They should be unique among siblings and stable over time. When a list is re-rendered, React uses the keys to identify which items have changed, been added, or been removed.

The key prop in the example above assigns a unique key to each list item based on its index. However, it's often better to use a unique identifier from your data (e.g., an id field) as the key.

```
function NameList(props) {
  const names = props.names;
  const nameListItems = names.map((name) => (
    <li key={name.id}>{name.name}</li>
  ));

  return <ul>{nameListItems}</ul>;
}
```

In this improved example, we use a unique id as the key for each list item.

Dynamically Rendering Lists

You can dynamically render lists based on conditions or data. For example, showing only even numbers from an array:

```
function EvenNumberList(props) {
  const numbers = props.numbers;
  const evenNumbers = numbers.filter((number) => number % 2 ===
0);
  const numberListItems = evenNumbers.map((number, index) => (
    <li key={index}>{number}</li>
  ));

  return <ul>{numberListItems}</ul>;
```

```
}
```

In this example, we use the filter method to create a new array containing only even numbers, and then we render them in a list.

Conditional Rendering within Lists

You can use conditional rendering within lists to display specific content for each item based on data. For instance, rendering a list of users with their status:

```
function UserList(props) {
  const users = props.users;
  const userListItems = users.map((user) => (
    <li key={user.id}>
      {user.name} - {user.isActive ? "Active" : "Inactive"}
    </li>
  ));

  return <ul>{userListItems}</ul>;
}
```

In this example, the user's status (active or inactive) is conditionally displayed for each user in the list.

React's ability to efficiently update and re-render lists based on keys is essential for optimizing performance.

In the next chapter, we will explore the concept of forms in React, including controlled and uncontrolled components, and how to handle user input.

Chapter 9: Routing with react-router-dom

Routing is a crucial aspect of building single-page applications (SPAs) using React. It allows you to navigate between different views or pages within your application while maintaining a smooth user experience. React doesn't come with built-in routing capabilities, but you can easily implement routing using third-party libraries. One of the most popular libraries for this purpose is react-router.

Here's how you can set up routing in a React application using react-router.

Step 1: Installation

First, you need to install react-router-dom, which is the routing library for React.

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

Step 2: Create Route Components

In your application, you need to define the components for each route or view. These components represent the content that will be displayed when a particular route is accessed.

```
// Home.js
import React from 'react';

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

export default Home;


// About.js
import React from 'react';
```

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

export default About;
```

Step 3: Set Up Routing

In your main application file, you should set up the routing using the components provided by *react-router-dom*, such as BrowserRouter, Route, and Switch.

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-
router-dom';
import Home from './Home';
import About from './About';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" exact component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Router>
  );
}

export default App;
```

In this example, we have two routes: the home page at the root path (/) and the about page at /about.

Step 4: Navigation Links

You can create navigation links using the Link component provided by react-router-dom. These links enable users to navigate between different views.

```
// MenuList.js
import React from 'react';
import { Link } from 'react-router-dom';
```

```
function MenuList() {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
      </ul>
    </nav>
  );
}
export default MenuList;
```

Include this MenuList component in your main application to provide MenuList links to your users.

Step 5: Run Your Application

Now, when you run your application, you'll be able to navigate between the different routes using the navigation links. The content of the corresponding route components will be displayed when you click on the links.

That's the basic setup for routing in a React application using react-router. You can expand upon this foundation to build more complex navigation and routing structures as needed for your project.

Chapter 10: Forms in React

Forms are a critical part of web applications, allowing users to input data, submit information, and interact with the application. In React, handling forms involves creating controlled components to manage form input and maintain the form's state. In this chapter, we'll dive into how to work with forms in React.

Controlled Components

In React, controlled components are a way to manage form elements, such as text inputs and checkboxes, by controlling their state through React state variables. This means React has full control over the form elements and their values.

Text Input Example

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

function ControlledForm() {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <form>
      <input
        type="text"
```

```
        value={inputValue}
        onChange={handleChange}
      />
      <p>You typed: {inputValue}</p>
    </form>
  );
}
```

In this example, the value of the input field is controlled by the inputValue state variable, and the onChange event handler updates the state whenever the user types in the input.

Checkbox Example

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

function ControlledCheckbox() {
  const [isChecked, setIsChecked] = useState(false);

  const handleCheckboxChange = (event) => {
    setIsChecked(event.target.checked);
  };

  return (
    <form>
      <label>
        <input
          type="checkbox"
          checked={isChecked}
          onChange={handleCheckboxChange}
        />
        Check this box
      </label>
      <p>Is checked: {isChecked ? 'Yes' : 'No'}</p>
    </form>
  );
```

```
}
```

In this example, the checked state of the checkbox is controlled by the isChecked state variable, and the onChange event handler updates the state when the checkbox is checked or unchecked.

Uncontrolled Components

Uncontrolled components allow form elements to maintain their own state, which is not controlled by React. These components can be useful for integrating React with non-React code or libraries. However, controlled components are generally recommended because they provide a more predictable and consistent user experience.

Form Submission

Handling form submission in React involves using the onSubmit event handler on the <form> element. You can prevent the default form submission behavior and perform custom actions, such as sending data to a server or updating the UI.

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

function FormSubmission() {
  const [inputValue, setInputValue] = useState('');
  const [submittedValue, setSubmittedValue] = useState('');

  const handleSubmit = (event) => {
    event.preventDefault();
    setSubmittedValue(inputValue);
  };

  return (
```

```
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={inputValue}
        onChange={(event) => setInputValue(event.target.value)}
      />
      <button type="submit">Submit</button>
      <p>Submitted value: {submittedValue}</p>
    </form>
  );
}
```

In this example, the onSubmit event handler prevents the default form submission, and the submitted value is displayed below the form.

Handling forms in React involves creating controlled components for form elements, which allows React to manage their state and behavior. Controlled components ensure a predictable and consistent user experience, making it easier to validate, submit, and interact with user input.

=====

Controlled Components and Uncontrolled Components in React

In React, when working with forms and user input, you have two main approaches: controlled components and uncontrolled components. Each has its own use cases and benefits, and choosing the right approach depends on your application's requirements. In this chapter, we'll delve into controlled and uncontrolled components in React.

Controlled Components

Controlled components are React components in which the form elements, such as text inputs and checkboxes, are controlled by the component's state. In other words, the component maintains the value of the form elements and updates them as the state changes.

Here's an example of a controlled component:

```
import React, { Component } from 'react';

class ControlledForm extends Component {
  constructor(props) {
    super(props);
    this.state = { inputValue: '' };
  }

  handleInputChange = (event) => {
    this.setState({ inputValue: event.target.value });
  };

  render() {
    return (
      <form>
        <input
          type="text"
          value={this.state.inputValue}
          onChange={this.handleInputChange}
        />
        <p>You typed: {this.state.inputValue}</p>
      </form>
    );
  }
}
```

In this example, the inputValue state variable controls the value of the text input. The onChange event handler updates the state when the user types in the input field, and the input value is always in sync with the component's state.

Uncontrolled Components

Uncontrolled components are React components where the form elements are not controlled by React's state but rather by the DOM. Uncontrolled components are typically used when you want to integrate React with non-React code or libraries or when you have a large number of form elements, and it's impractical to manage the state for each one.

Here's an example of an uncontrolled component:

```
import React, { useRef } from 'react';

function UncontrolledForm() {
  const inputRef = useRef();

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`You typed: ${inputRef.current.value}`);
  };

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

In this example, the inputRef is used to directly access the DOM element. The form elements are uncontrolled, and the value is retrieved using the inputRef.current.-value when the form is submitted.

Choosing Between Controlled and Uncontrolled Components

When deciding whether to use controlled or uncontrolled components, consider the following:

Controlled Components: Use controlled components when you need to maintain precise control over the form elements, validate user input, and respond to changes in real-time. Controlled components are especially useful for forms that require complex validation and dynamic interactions.

Uncontrolled Components: Use uncontrolled components when integrating with third-party libraries, working with a large number of form elements, or when the input values are not critical to your application's state management. Uncontrolled components provide a simpler way to interact with the DOM.

Controlled and uncontrolled components in React provide different approaches to managing form elements and user input. Choosing the right approach depends on your specific use case and application requirements. Whether you need fine-grained control over form elements or a simpler way to work with the DOM, React offers flexibility to meet your needs.

=====

In the next chapter, we will explore more advanced React concepts, including React Hooks, custom hooks, code-splitting, and build and deployment strategies for React applications.

Chapter 11: The useEffect Hook and API Calls

The useEffect hook is a fundamental part of React for managing side effects, including making API calls. When working with API calls, it's crucial to handle loading, success, and error states to provide a better user experience. In this chapter, we'll explore how to use the useEffect hook to make API calls while managing these states.

The useEffect Hook

The useEffect hook is used to manage side effects in React components. It can be used to make API calls, among other things. It takes two arguments: a function that contains the code to run and an array of dependencies.

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

function Example() {
  const [loading, setLoading] = useState(true);
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    // Code for side effect (e.g., API call)
  }, []);

  return (
    <div>
      {/* Render UI based on loading, data, and error states */}
    </div>
```

```
  );
}
```

The loading, data, and error states will help us manage the loading, success, and error states of the API call.

Making an API Call with Loading, Success, and Error States

When making an API call, you can manage the loading, success, and error states to provide a better user experience. Here's an example of how to structure your code to handle these states:

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

function DataFetchingExample() {
  const [loading, setLoading] = useState(true);
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then((response) => {
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        return response.json();
      })
      .then((result) => {
        setData(result);
        setLoading(false);
      })
      .catch((err) => {
        setError(err);
        setLoading(false);
      });
```

```
  }, []);

  return (
    <div>
      {loading ? (
        <p>Loading...</p>
      ) : error ? (
        <p>Error: {error.message}</p>
      ) : (
        <p>Data: {data}</p>
      )}
    </div>
  );
}
```

In this example, we initially set loading to true. When the API call is in progress, loading is true, and the "Loading..." message is displayed. If the call is successful, loading is set to false, and the fetched data is displayed. If there is an error, loading is set to false, and the error message is displayed.

In the next few chapters, we will explore more advanced React topics, such as HOC, Error Boundary, custom hooks, code-splitting, and build and deployment strategies for React applications.

Chapter 12: Higher-Order Components with Functional Components

Higher-Order Components (HOCs) are a powerful design pattern in React that allow you to reuse component logic and share it across different parts of your application. While HOCs were traditionally implemented with class components, they can also be applied to functional components using React hooks and the new patterns introduced in modern React. In this chapter, we'll delve into how to use HOCs with functional components.

What Are Higher-Order Components?

Higher-Order Components are functions that take a component and return a new component with enhanced features. HOCs are used for cross-cutting concerns like logging, authentication, and data fetching. They allow you to separate concerns and create more reusable and maintainable code.

Example of a Higher-Order Component

Let's consider a simple example of a HOC that logs the time it took to render a component:

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

const withRenderTimeLogging = (WrappedComponent) => {
  return function WithRenderTimeLogging(props) {
    const start = performance.now();
    useEffect(() => {
      const end = performance.now();
      console.log(`Rendered ${WrappedComponent.name} in ${end -
start}ms`);
    });

    return <WrappedComponent {...props} />;
  };
};
```

In this example, withRenderTimeLogging is a HOC that logs the render time of a component it wraps. It takes a WrappedComponent as an argument, enhances it with the render time logging, and returns a new component.

Using a Higher-Order Component

To use a HOC, you wrap a component with it. Here's an example of using the with-RenderTimeLogging HOC:

```
const MyComponent = ({ text }) => {
  return <div>{text}</div>;
};

const MyComponentWithLogging =
withRenderTimeLogging(MyComponent);

function App() {
  return (
    <div>
      <MyComponentWithLogging text="Hello, World!" />
    </div>
  );
```

```
}
```

In this example, MyComponentWithLogging is a new component created by wrapping MyComponent with the withRenderTimeLogging HOC. When MyComponentWithLogging is rendered, it logs the time it took to render MyComponent.

Benefits of HOCs with Functional Components

Using functional components with HOCs offers several benefits:

Composition: You can compose multiple HOCs to add different functionalities to a component, promoting code reuse.

Readability: Functional components with HOCs are often more concise and easier to read than class components with HOCs.

Hooks Compatibility: HOCs can seamlessly work with React hooks, allowing you to combine different approaches to state management and logic reuse.

Higher-Order Components provide a powerful way to enhance the functionality and behavior of your components. By adapting the HOC pattern to functional components, you can create more readable and reusable code while benefiting from the flexibility of React hooks.

In the next chapter, we'll explore the concept of error boundaries in React and how they can help handle errors in your application.

Chapter 13: Fragments in React

Fragments in React provide a way to group multiple elements without adding extra nodes to the DOM. They are especially useful when you need to return multiple elements from a component's render method. In this chapter, we'll delve into how to use Fragments in React.

What Are Fragments?

In React, when you return multiple elements from a component's render method, they must be wrapped in a single parent element. This requirement is due to the way React reconciles the virtual DOM. However, sometimes you don't want to introduce an extra element in the actual DOM, which is where Fragments come into play.

Using Fragments

Fragments in React are represented using an empty angle bracket syntax or the <> and </> tags. You can use them to group multiple elements without adding any extra nodes to the DOM. Here's an example:

```
import React from 'react';

function MyComponent() {
```

```
  return (
    <>
      <h1>Title</h1>
      <p>Paragraph 1</p>
      <p>Paragraph 2</p>
    </>
  );
}
```

In this example, the <> and </> tags are used to create a fragment that wraps the h1 and two p elements. These elements are grouped together without introducing an additional DOM node.

Fragments and Keys

If you need to map over an array and render multiple elements, you can use Fragments with keys to avoid the common "Each child in a list should have a unique 'key'" warning. Here's an example:

```
import React from 'react';

function ListItems({ items }) {
  return (
    <>
      {items.map((item) => (
        <React.Fragment key={item.id}>
          <p>{item.name}</p>
        </React.Fragment>
      ))}
    </>
  );
}
```

In this example, the key is applied to each React.Fragment element, ensuring that each one has a unique identifier.

Benefits of Fragments

Fragments provide several benefits in React:

Cleaner Code: Fragments allow you to group elements without introducing unnecessary parent elements in the DOM, resulting in cleaner and more semantic code.

No Extra Styling: Since fragments don't create additional DOM nodes, you won't need to account for extra styling or layout considerations in your CSS.

Avoiding Warning: When mapping over elements and using Fragments with keys, you can avoid the common React warning regarding missing or non-unique keys.

Fragments in React are a useful feature for grouping multiple elements without adding extra nodes to the DOM. They provide cleaner code and help avoid common issues related to element grouping and keys.

In the next chapter, we'll explore the concept of error boundaries in React and how they can help handle errors in your application.

Chapter 14: Error Boundaries in React

Error boundaries in React are a way to gracefully handle and recover from errors that occur within a component's tree. They are an essential part of building reliable and user-friendly applications. In this chapter, we'll delve into how to use error boundaries in React.

What Are Error Boundaries?

In React, error boundaries are special components that catch JavaScript errors anywhere in their child component tree, log them, and display a fallback UI instead of crashing the whole application. Error boundaries allow you to separate the error-handling code from the regular UI components.

Creating an Error Boundary

To create an error boundary in React, you need to define a class component with two special methods: componentDidCatch and render.

Important Note: As per React JS Version 18 documentation, class components are not recommended any more. There is currently no way to write an error boundary as a function component. However, you don’t have to write the error boundary class yourself. For example, you can use **react-error-boundary** package instead.

Steps to use react-error-boundary package

You can use the react-error-boundary package to catch and handle errors in a more user-friendly and customizable way. This package simplifies the process of implementing error boundaries in your React application. Here's how you can catch errors using the react-error-boundary package.

1. Installing react-error-boundary

To get started, you'll need to install the react-error-boundary package:

```
npm install react-error-boundary
```

or

```
yarn add react-error-boundary
```

2. Using react-error-boundary to Catch Errors

Once you've installed the package, you can create an error boundary component using the ErrorBoundary component provided by react-error-boundary. This component simplifies the process of catching and handling errors.

Here's an example of how to use react-error-boundary:

```
import React from 'react';
import { ErrorBoundary } from 'react-error-boundary';

function MyComponent() {
  // Simulate an error
  if (/* some condition */) {
    throw new Error('An error occurred!');
  }

  return <p>No error occurred.</p>;
}
```

```
// this is a component that serves fallback UI
function ErrorFallbackUI({ error, resetErrorBoundary }) {
  return (
    <div>
      <p>Something went wrong:</p>
      <pre>{error.message}</pre>
      <button onClick={resetErrorBoundary}>Try again</button>
    </div>
  );
}

// When MyComponent has an error -- fallback UI will show up
function App() {
  return (
    <div>
      <ErrorBoundary FallbackComponent={ErrorFallbackUI}
onReset={() => window.location.reload()}>
        <MyComponent />
      </ErrorBoundary>
    </div>
  );
}
```

In this example, the ErrorBoundary component wraps MyComponent. When an error is thrown within MyComponent, it's caught by the ErrorBoundary. The ErrorFallbackUI component is used to display a user-friendly error message, and a "Try again" button is provided for recovery.

The react-error-boundary package simplifies the implementation of error boundaries in React, making it easier to catch and handle errors gracefully. By customizing the error handling components, you can provide a better user experience and maintain the reliability of your application.

Customizing Error Handling

You can customize the error handling by providing your own ErrorFallbackUI component and specifying the behavior of the onReset callback. This allows you to control how errors are presented and how to recover from them.

Benefits of Error Boundaries

Error boundaries in React provide several benefits:

Error Isolation: Errors are contained within the error boundary, preventing them from affecting the entire application.

Fallback UI: Error boundaries allow you to display a user-friendly error message or fallback UI when an error occurs.

Improved Reliability: Using error boundaries helps ensure that your application remains reliable even in the presence of errors.

Prevents Application Crashes: Errors caught by the error boundary won't crash the entire application, ensuring a smoother user experience.

Error boundaries in React are a crucial tool for handling errors gracefully and maintaining the stability of your application. They enable you to catch and handle errors within a specific component tree while providing a better user experience.

In the next chapter, we'll explore the concept of sharing data between random components using one of the advanced hooks in React.

Chapter 15: Sharing Data Between Components In React

Sharing data between components in React is a common task and can be accomplished using various techniques, depending on the component hierarchy and complexity of your application. Here are some approaches for sharing data between components in React:

1. Props: The simplest way to share data between components is by passing it as props. Parent components can pass data down to child components as props. This is suitable for sharing data in a one-way flow from parent to child.

Example:

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const data = 'Hello from Parent';
  return <ChildComponent data={data} />;
}


// ChildComponent.js
import React from 'react;

function ChildComponent(props) {
  return <div>{props.data}</div>;
}
```

2. Context API: The Context API allows you to share data across components without explicitly passing it through props at every level. It's especially useful for global state management.

Example:

```
// DataContext.js
import React, { createContext, useContext, useState } from
'react';

const DataContext = createContext();

export function DataProvider({ children }) {
  const [data, setData] = useState('Global Data');

  return (
    <DataContext.Provider value={{ data, setData }}>
      {children}
    </DataContext.Provider>
  );
}

export function useData() {
  return useContext(DataContext);
}
```

Usage:

```
// ParentComponent.js
import React from 'react';
import { useData } from './DataContext';

function ParentComponent() {
  const { data } = useData();
  return <div>{data}</div>;
}
```

3. State Management Libraries: If your application involves complex state management, you can use state management libraries like Redux or Mobx to share and

manage data across components. These libraries provide centralized stores and actions for state management.

4. Event Emitter or Pub-Sub Pattern: You can implement a custom event emitter or pub-sub pattern to allow components to subscribe to and publish events. This enables communication between components that are not directly connected in the component tree.

Example:

```
// EventEmitter.js
class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(event, listener) {
    if (!this.events[event]) {
      this.events[event] = [];
    }
    this.events[event].push(listener);
  }

  emit(event, data) {
    if (this.events[event]) {
      this.events[event].forEach(listener => listener(data));
    }
  }
}

const eventEmitter = new EventEmitter();
export default eventEmitter;
```

Usage:

```
// ComponentA.js
import React from 'react';
import eventEmitter from './EventEmitter';

function ComponentA() {
```

```
  eventEmitter.on('customEvent', data => {
    console.log('Component A received:', data);
  });

  return <div>Component A</div>;
}


// ComponentB.js
import React from 'react';
import eventEmitter from './EventEmitter';

function ComponentB() {
  eventEmitter.emit('customEvent', 'Hello from Component B');
  return <div>Component B</div>;
}
```

5. Router Props: When working with React Router, you can pass data between components using route props. This allows you to send data as part of the URL or route parameters.

Example:

```
// ParentComponent.js
import { Link } from 'react-router-dom';

function ParentComponent() {
  const data = 'Data for Child';
  return (
    <Link to={{ pathname: '/child', state: { data } }}>Go to
Child</Link>
  );
}


// ChildComponent.js
import { useLocation } from 'react-router-dom';

function ChildComponent() {
  const location = useLocation();
  const { data } = location.state;
```

```
  return <div>{data}</div>;
}
```

These are some of the ways to share data between components in React. The choice of method depends on your application's architecture and the specific requirements for data sharing.

Chapter 16: Hooks, Rules of Hooks, Advanced Hooks and Custom Hook

In this chapter, we'll deepen our understanding of React Hooks and the rules associated with them. We'll also explore advanced Hooks and apply these concepts to create a product listing and an add to cart feature in a React application. Our plan is to share data between different components in React JS.

React Hooks and Rules of Hooks

React Hooks are a vital part of modern React development, allowing functional components to manage state and side effects. As mentioned previously, the "Rules of Hooks" are essential to ensure proper usage of Hooks:

1. Only Call Hooks at the Top Level: Hooks should be called at the top level of your functional components, not inside loops, conditions, or nested functions.
2. Only Call Hooks from React Functions: You should call Hooks from within React functional components. Custom Hooks should also adhere to this rule.

Advanced Hooks

React provides several advanced built-in Hooks, including useContext and useReducer. We'll use these Hooks to build a product listing and an add to cart feature.

Context API with useContext Hook

The useContext Hook allows you to access context values, making it ideal for managing global state. In our scenario, we'll use it to share product data across components.

State Management with useReducer Hook

The useReducer Hook is a suitable choice for managing complex state logic. In our application, we'll use it to handle the shopping cart state.

Building a Product Listing and Add to Cart Feature

Let's create a simplified product listing and add to cart feature using React Hooks. We'll leverage the Context API with useContext and useReducer for state management.

Here's a basic structure of our application:

```
// ProductList.js
import React, { useContext } from 'react';
```

```
import { ProductContext } from './ProductContext';

function ProductList() {
  const { products, addToCart } = useContext(ProductContext);

  return (
    <div>
      <h2>Product List</h2>
      <ul>
        {products.map((product) => (
          <li key={product.id}>
            {product.name}
            <button onClick={() => addToCart(product)}>Add to
Cart</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
export default ProductList;


// ShoppingCart.js
import React, { useContext } from 'react';
import { ProductContext } from './ProductContext';

function ShoppingCart() {
  const { cart } = useContext(ProductContext);

  return (
    <div>
      <h2>Shopping Cart</h2>
      <ul>
        {cart.map((item) => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default ShoppingCart;


// ProductContext.js
import React, { createContext, useReducer } from 'react';

const initialState = {
  products: [
    { id: 1, name: 'Product A' },
    { id: 2, name: 'Product B' },
    // Add more products
  ],
  cart: [],
};

const productReducer = (state, action) => {
  switch (action.type) {
    case 'ADD_TO_CART':
      return { ...state, cart: [...state.cart, action.payload] };
    default:
      return state;
  }
};

export const ProductContext = createContext();

export const ProductProvider = ({ children }) => {
  const [state, dispatch] = useReducer(productReducer,
initialState);

  const addToCart = (product) => {
    dispatch({ type: 'ADD_TO_CART', payload: product });
  };

  return (
    <ProductContext.Provider value={{ ...state, addToCart }}>
      {children}
    </ProductContext.Provider>
  );
};


// App.js
```

```
import React from 'react';
import ProductList from './ProductList';
import ShoppingCart from './ShoppingCart';
import { ProductProvider } from './ProductContext';

function App() {
  return (
    <ProductProvider>
      <div>
        <ProductList />
        <ShoppingCart />
      </div>
    </ProductProvider>
  );
}

export default App;
```

In this example, we have created a product listing component and a shopping cart component. The state, including products and the shopping cart, is managed using the Context API with useContext and useReducer. When the "Add to Cart" button is clicked, the addToCart function is called to update the shopping cart state.

React Hooks are a powerful feature that simplifies state management and component logic in functional components. By following the rules of Hooks and leveraging advanced Hooks like useContext and useReducer, you can build complex applications while keeping your code clean and maintainable.

Custom Hooks

Custom Hooks are a powerful and reusable way to share logic between React components. They allow you to extract stateful logic and side effects into a separate function, making your code more modular and easier to maintain. In this example,

I'll show you how to create a custom hook to manage a simple counter that can be shared across components.

Step 1: Create a Custom Hook

Create a new file for your custom hook, for example, useCounter.js. This file will contain the logic for the counter.

```
// useCounter.js
import { useState } from 'react';

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

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  const reset = () => {
    setCount(initialValue);
  };

  return { count, increment, decrement, reset };
}

export default useCounter;
```

In this custom hook, we're using the useState hook to manage the state of the counter and providing functions to increment, decrement, and reset the counter.

Step 2: Using the Custom Hook in Components

Now, you can use the custom hook in your components to share the counter logic. Here's an example of how to use it in two different components.

```
// ComponentA.js
import React from 'react';
```

```
import useCounter from './useCounter';

function ComponentA() {
  const { count, increment, decrement, reset } = useCounter(0);

  return (
    <div>
      <h2>Component A</h2>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}

export default ComponentA;


// ComponentB.js
import React from 'react';
import useCounter from './useCounter';

function ComponentB() {
  const { count, increment, decrement, reset } = useCounter(100);

  return (
    <div>
      <h2>Component B</h2>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}

export default ComponentB;
```

Both ComponentA and ComponentB are using the useCounter custom hook to manage their individual counter states.

Step 3: Render the Components in Your App

To see the custom hook in action, render ComponentA and ComponentB in your main application component.

```
// App.js
import React from 'react';
import ComponentA from './ComponentA';
import ComponentB from './ComponentB';

function App() {
  return (
    <div>
      <ComponentA />
      <ComponentB />
    </div>
  );
}

export default App;
```

Step 4: Run Your Application

You can now run your application using *npm start* and see how both ComponentA and ComponentB share the same counter logic provided by the useCounter custom hook.

Custom hooks provide a clean and reusable way to share stateful logic across components, improving code maintainability and reusability. You can create custom hooks for various purposes, including data fetching, form handling, and more, depending on the needs of your application.

Chapter 17: Building more complex navigation and routing structures

To build more complex navigation and routing structures in your React application using react-router, you can follow these advanced techniques and best practices. These will help you manage a larger number of routes and create a more dynamic navigation experience:

Nested Routes: Create nested routes to structure your application and handle sub-routes within specific sections. For example, you might have a dashboard with sub-pages for different modules or features.

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-
router-dom';
import Home from './Home';
import Dashboard from './Dashboard';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" exact component={Home} />
        <Route path="/dashboard" component={Dashboard} />
      </Switch>
    </Router>
  );
}
```

In the Dashboard component, you can define additional routes specific to the dashboard.

Dynamic Routes: Use route parameters to create dynamic routes that change based on user input or data. For example, you might have routes for displaying individual user profiles.

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-
router-dom';
import Home from './Home';
import UserProfile from './UserProfile';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" exact component={Home} />
        <Route path="/user/:id" component={UserProfile} />
      </Switch>
    </Router>
  );
}
```

In the UserProfile component, you can access the id parameter to load user-specific data.

Protected Routes: Implement authentication and authorization for protected routes. You can create a higher-order component (HOC) or a custom PrivateRoute component that checks if a user is authenticated before rendering a protected route.

```
// PrivateRoute.js
import React from 'react';
import { Route, Redirect } from 'react-router-dom';

const PrivateRoute = ({ component: Component,
isAuthenticated, ...rest }) => (
```

```
  <Route
    {...rest}
    render={(props) =>
      isAuthenticated ? <Component {...props} /> : <Redirect
to="/login" />
    }
  />
);

export default PrivateRoute;
```

In your main app, you can use the PrivateRoute to protect specific routes:

```
// App.js
import React, { useState } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-
router-dom';
import Home from './Home';
import UserProfile from './UserProfile';
import PrivateRoute from './PrivateRoute';

function App() {
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  return (
    <Router>
      <Switch>
        <Route path="/" exact component={Home} />
        <Route path="/user/:id" component={UserProfile} />
        <PrivateRoute
          path="/protected"
          component={ProtectedComponent}
          isAuthenticated={isAuthenticated}
        />
      </Switch>
    </Router>
  );
}
```

Lazy Loading and Code Splitting: To optimize performance, use code splitting to load only the necessary components and dependencies for each route. This can be achieved using tools like Webpack and dynamic imports.

```
// Load a component lazily
const LazyComponent = React.lazy(() => import('./
LazyComponent'));

// Route with lazy loading
<Route
  path="/lazy"
  render={(props) => (
    <React.Suspense fallback={<div>Loading...</div>}>
      <LazyComponent {...props} />
    </React.Suspense>
  )}
/>
```

Route Guards: Implement route guards for additional navigation logic, such as checking if a user can navigate to a certain route, confirming unsaved changes, or handling user permissions.

Navigation Menus: Create dynamic navigation menus that change based on the current route and user roles. You can use the NavLink component for active route styling. NavLink will ensure you get active menu highlighting when you click on the navigation menu.

```
// MenuList
import React from 'react';
import { NavLink } from 'react-router-dom';

function MenuList() {
  return (
    <nav>
      <ul>
        <li>
          <NavLink exact to="/">Home</NavLink>
        </li>
```

```
          <li>
            <NavLink to="/dashboard">Dashboard</NavLink>
          </li>
          <li>
            <NavLink to="/user/123">User Profile</NavLink>
          </li>
        </ul>
      </nav>
    );
}
```

By following these advanced techniques and best practices, you can build complex navigation and routing structures in your React application while ensuring a smooth and user-friendly experience. It's essential to consider your application's specific requirements and architecture when implementing these strategies.

Chapter 18: Authentication in React JS

Implementing authentication in a React application is a crucial aspect of building secure web applications. Authentication ensures that only authorized users can access certain parts of your application or perform specific actions. There are several methods for implementing authentication in a React application, and the choice depends on your project requirements. Here's a general outline of the steps involved:

1. User Authentication Service: You need a backend service to handle user authentication. This service should provide endpoints for user registration, login, and logout. Popular backend technologies for building this service include Node.js with Express, Ruby on Rails, Django, and more. You can use third-party authentication providers like Firebase, Auth0, or OAuth for a quicker setup.
2. User Registration: Implement a registration form in your React application where users can create new accounts. When a user registers, their information (e.g., username and password) is sent to the backend service for storage. You should also consider email verification and CAPTCHA to prevent abuse.
3. User Login: Create a login form that allows users to enter their credentials and submit them to the authentication service for verification. If the credentials are correct, the user should receive an authentication token,

typically a JSON Web Token (JWT), which is used to identify and authenticate the user in subsequent requests.

4. Token Management: Store the JWT token securely on the client side. You can use browser storage options like localStorage or sessionStorage, or modern solutions like HTTP-only cookies with the SameSite attribute set to "strict" to prevent cross-site request forgery (CSRF) attacks. Be mindful of security concerns when managing tokens.
5. Authentication Context: Implement an authentication context in your React application using the Context API or a state management library like Redux or Mobx. This context should provide state and methods for checking if a user is authenticated, storing user information, and handling login and logout actions.
6. Protecting Routes: Use React Router to define protected routes that require authentication. Create a higher-order component (HOC) or a custom PrivateRoute component to check if the user is authenticated. If not, redirect them to the login page.

```
// PrivateRoute.js
import React, { useContext } from 'react';
import { Route, Redirect } from 'react-router-dom';
import { AuthContext } from './AuthContext';

function PrivateRoute({ component: Component, ...rest }) {
  const { isAuthenticated } = useContext(AuthContext);

  return (
    <Route
      {...rest}
      render={(props) =>
        isAuthenticated ? <Component {...props} /> : <Redirect
to="/login" />
      }
    />
```

```
  );
}

export default PrivateRoute;
```

7. User Profile: Create a user profile page where authenticated users can view and edit their profile information. This page should make authenticated requests to the backend service to fetch and update user data.
8. Logout: Implement a logout functionality that clears the user's session by removing the authentication token from storage. Make sure to handle any logout-related clean-up on the server-side as well.
9. Error Handling: Handle authentication-related errors gracefully. Show appropriate error messages to the user in case of failed login attempts, expired tokens, or other authentication issues.
10. Security: Be aware of security best practices, such as protecting against cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks, securing API endpoints, and using HTTPS for secure data transfer.
11. Remember Me: Implement a "Remember Me" feature, allowing users to stay logged in across sessions, and provide options for session timeout.
12. Multi-factor Authentication (MFA): Consider adding MFA for an extra layer of security, which can be in the form of SMS codes, email verification, or authenticator apps.
13. Password Reset: Enable users to reset their passwords through a secure process, such as sending a reset link to their email.
14. Access Control: Implement role-based access control (RBAC) or permissions to manage what users can do within your application. Some users might have different levels of access.

15. Logging and Auditing: Log authentication and authorization events for auditing and troubleshooting. Monitoring failed login attempts and suspicious activities can help enhance security.
16. Testing: Thoroughly test your authentication system, including edge cases, vulnerabilities, and error handling. Automated and manual testing can help identify and fix issues.

Keep in mind that securing your application requires continuous attention to security best practices, as new threats and vulnerabilities emerge. Be sure to stay updated on the latest security recommendations and consider regular security assessments.

Implement Authentication in React JS

Implementing authentication in a React application typically involves a server-side component (e.g., a backend service) for user registration, login, and token generation. In this example, I'll show you how to integrate authentication in a simple React application using the Context API and a fake authentication service.

Step 1: Create an Authentication Context

Create an AuthContext.js file to define the authentication context using the React Context API.

```
// AuthContext.js
import React, { createContext, useContext, useState, useEffect }
from 'react';

const AuthContext = createContext();

export const useAuth = () => {
  return useContext(AuthContext);
};
```

```
export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  // Simulate a user login by setting user state if a token
exists in localStorage
  useEffect(() => {
    const token = localStorage.getItem('token');
    if (token) {
      // You can send a request to the server to verify the
token's authenticity
      setUser({ username: 'sampleUser' });
    }
  }, []);

  const login = (token) => {
    localStorage.setItem('token', token);
    setUser({ username: 'sampleUser' });
  };

  const logout = () => {
    localStorage.removeItem('token');
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};
```

Step 2: Create Login and Logout Components

You can create login and logout components for your application. In a real-world scenario, these would be full-fledged forms and views, but for the sake of simplicity, we'll create basic components.

```
// Login.js
import React from 'react';
import { useAuth } from './AuthContext';
```

```
function Login() {
  const { login } = useAuth();

  const handleLogin = () => {
    // In a real application, this is where you would make a
request to your backend for authentication.
    // Here, we're simulating a successful login by setting a
dummy token.
    login('sample-token');
  };

  return (
    <div>
      <h2>Login</h2>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}

export default Login;


// Logout.js
import React from 'react';
import { useAuth } from './AuthContext';

function Logout() {
  const { logout } = useAuth();

  const handleLogout = () => {
    // In a real application, you would send a request to your
backend to invalidate the token.
    logout();
  };

  return (
    <div>
      <h2>Logout</h2>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
```

```
}

export default Logout;
```

Step 3: Set Up Protected Routes

You can create protected routes that require authentication using the react-router-dom library. We'll define a simple route for protected content that can only be accessed if the user is logged in.

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-
router-dom';
import { AuthProvider } from './AuthContext';
import Login from './Login';
import Logout from './Logout';

function ProtectedContent() {
  return <div>Protected Content</div>;
}

function App() {
  return (
    <AuthProvider>
      <Router>
        <Switch>
          <Route path="/login" component={Login} />
          <Route path="/logout" component={Logout} />
          <Route path="/protected" component={ProtectedContent} /
>
        </Switch>
      </Router>
    </AuthProvider>
  );
}

export default App;
```

Step 5: Testing the Authentication

You can now test your authentication system by running the application and navigating between login, logout, and protected routes.

To start the development server, run:

```
npm start
```

Open your browser and access the following URLs:

http://localhost:3000/login: This is where you can simulate a login.

http://localhost:3000/logout: Simulate a logout.

http://localhost:3000/protected: Access the protected route, which can only be viewed if you're logged in.

In this example, we've created a basic authentication system using React's Context API. In a real-world application, you would replace the dummy token and user verification with a server-based authentication system. Additionally, you should handle error cases and add security measures for storing tokens more securely.

Remember that this example is for educational purposes and doesn't include proper security practices for a production application. Proper security measures, such as token validation, token expiration, and more, should be implemented when building a real authentication system.

Chapter 19: Code splitting

Code splitting is a technique used in React applications to improve performance by splitting the application's bundle into smaller chunks, allowing for more efficient loading of code on-demand. Code splitting reduces the initial load time of your application, which is especially important for larger apps. In React, you can implement code splitting using the React.lazy function and the Suspense component, which are part of React's core. This will be often times referred to as lazy loading of components.

Here's how to implement code splitting in a React application:

Step 1: Install React and React Router (if not already installed)

Ensure you have React and React Router installed in your project. You can install them using npm or yarn:

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

Step 2: Create a Lazy-Loaded Component

First, create a component that you want to lazy load. For example, let's create a component called ContactUs:

```
// ContactUs.js
import React from 'react';

function ContactUs() {
  return <div>Contact Us</div>;
}

export default ContactUs;
```

Step 3: Implement Code Splitting

In your main component, you can use the React.lazy function to dynamically load the ContactUs when it's needed. Wrap the lazy-loaded component in a Suspense component to display a loading indicator while the component is being loaded.

```
// App.js
import React, { Suspense } from 'react';
import { BrowserRouter as Router, Route, Link } from 'react-
router-dom';

const ContactUs = React.lazy(() => import('./ContactUs'));

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/contact-us">Contact Us</Link>
            </li>
          </ul>
        </nav>

        <hr />

        <Route path="/" exact component={Home} />
        <Route path="/contact-us" render={() => (
          <Suspense fallback={<div>Loading...</div>}>
            <ContactUs />
          </Suspense>
        )} />
      </div>
    </Router>
  );
}
```

```
function Home() {
  return <div>Home</div>;
}

export default App;
```

In this example, we import the ContactUs using React.lazy and provide a function that returns the dynamic import of the component. When the "/lazy" route is accessed, React will load the ContactUs only at that moment, showing the loading indicator specified in the Suspense component.

Step 4: Run Your Application

You can now run your application using npm start or a similar command, and when you navigate to the "/contact-us" route, you'll see the "Contact Us Component" being loaded on-demand.

Code splitting helps keep your initial bundle size smaller, improving load times for users, and optimizing your application's performance. You can apply code splitting to other parts of your application as needed, especially for large or less frequently used components.

Chapter 20: Internationalization and Localization

In the context of software internationalization and localization, "i18n" and "l10n" are abbreviations:

i18n (Internationalization):

The term "i18n" is short for "internationalization," and it refers to the process of designing and preparing your software so that it can be easily adapted to different languages and regions without engineering changes.

Internationalization involves making the software capable of handling various languages, character sets, date and time formats, and cultural conventions.

In the example provided, i18n is represented by the react-i18next library and the structure of language resources in different files (translation.json for English and French).

l10n (Localization):

The term "l10n" is short for "localization," and it refers to the process of adapting your internationalized software for a specific region or language by adding locale-specific components and translating text.

Localization involves providing translations for the user interface elements, such as labels, messages, and content, to make the software culturally and linguistically appropriate for a particular audience.

In the following example, l10n is demonstrated by the content inside the translation.json files, where the same keys have different values in English and French.

In summary, "i18n" is the broader process of making your software adaptable to various languages and regions, while "l10n" is the specific task of translating and adapting the content for a particular locale. Both are crucial for creating a user-friendly and globally accessible application. The combination of i18n and l10n allows developers to create applications that can be easily localized for different languages and cultures.

Example

I'll provide you with a simple example of a React component with internationalization (i18n) and localization (l10n) implemented using the react-i18next library. First, make sure to install the library:

```
npm install i18next react-i18next
```

Now, let's create a sample component with i18n and l10n:

```
// components/LocalizedComponent.js
import React from 'react';
import { useTranslation } from 'react-i18next';

const LocalizedComponent = () => {
  // The useTranslation hook provides the t function for
translation
  const { t, i18n } = useTranslation();

  // Function to change the language
  const changeLanguage = (language) => {
```

```
    i18n.changeLanguage(language);
  };

  return (
    <div>
      <h1>{t('welcome')}</h1>
      <p>{t('intro')}</p>

      <div>
        {/* Language selection buttons */}
        <button onClick={() => changeLanguage('en')}>English</
button>
        <button onClick={() => changeLanguage('fr')}>French</
button>
      </div>
    </div>
  );
};

export default LocalizedComponent;
```

Now, let's set up i18n in your main application file (e.g., App.js):

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-
router-dom';
import { initReactI18next } from 'react-i18next';
import i18n from 'i18next';
import LocalizedComponent from './components/LocalizedComponent';

// Import language files
import enTranslation from './locales/en/translation.json';
import frTranslation from './locales/fr/translation.json';

// Set up i18n with translations
i18n.use(initReactI18next).init({
  resources: {
    en: { translation: enTranslation },
    fr: { translation: frTranslation },
  },
  lng: 'en', // default language
  fallbackLng: 'en',
```

```
  interpolation: {
    escapeValue: false,
  },
});

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<LocalizedComponent />} />
      </Routes>
    </Router>
  );
};

export default App;
```

Now, create language files in locales directory:

```
// locales/en/translation.json
{
  "welcome": "Welcome to My App",
  "intro": "This is a sample localized component."
}


// locales/fr/translation.json
{
  "welcome": "Bienvenue sur mon application",
  "intro": "Ceci est un composant localisé d'exemple."
}
```

This is a basic example to get you started. You can expand on this by integrating with more advanced i18n features provided by react-i18next and organizing translations for various components. Additionally, you can explore features like **pluralization, date formatting, and more.**

creating a locales directory within the src folder is a common practice. It helps organize your localization files and keeps them within the source code structure. Here's an example directory structure:

```
/src
  /components
    LocalizedComponent.js
  /locales
    /en
      translation.json
    /fr
      translation.json
  App.js
```

- The locales directory contains subdirectories for each language (en for English, fr for French).
- Each language subdirectory contains a translation.json file with key-value pairs for translations.
- Feel free to adjust the structure based on your project's needs, but keeping localization files within the src folder is a good practice for clarity and maintainability.

How to set the French as default language?

To ensure that your app opens with content in French by default, you can set the default language for i18n when initializing it. In the example below, I'll show you how to modify the initialization of i18n in your App.js file to set French (fr) as the default language:

```
// App.tsx
// Set up i18n with translations
i18n.use(initReactI18next).init({
  resources: {
    en: { translation: enTranslation },
    fr: { translation: frTranslation },
```

```
  },
  lng: "fr", // Set French as the default language
  fallbackLng: "en",
  interpolation: {
    escapeValue: false,
  },
});
```

How to automatically set the language based on the user's location?

To automatically set the language based on the user's location, you can leverage a library like react-i18next along with the i18next-browser-languagedetector plugin, which detects the user's language based on the browser's language preferences.

Here are the steps to achieve this:

Install the required packages:

```
npm install i18next-browser-languagedetector
```

Modify your i18n initialization in App.js to include the language detector:

```
// App.js

import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-
router-dom';
import { initReactI18next } from 'react-i18next';
import i18n from 'i18next';
import LanguageDetector from 'i18next-browser-
languagedetector'; // Import the language detector
import LocalizedComponent from './components/LocalizedComponent';

// Import language files
import enTranslation from './locales/en/translation.json';
import frTranslation from './locales/fr/translation.json';
```

```
// Set up i18n with translations and language detector
i18n.use(initReactI18next)
  .use(LanguageDetector) // Use the language detector
  .init({
    resources: {
      en: { translation: enTranslation },
      fr: { translation: frTranslation },
    },
    fallbackLng: 'en',
    interpolation: {
      escapeValue: false,
    },
  });

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<LocalizedComponent />} />
      </Routes>
    </Router>
  );
};
export default App;
```

Now, when a user opens your site, i18next-browser-languagedetector will try to determine the user's language based on their browser's language preferences. If the detected language is available in your resources, it will be used; otherwise, it falls back to the default language ('en' in this case).

Keep in mind that this approach relies on the browser's language preferences, and it may not be accurate in all cases. Users can manually change the language using your language selection buttons if needed.

l10n Example

Now let's understand l10n. The language code for Australian English is 'en-AU'. Below is an example of how you might structure your Australian English localization (l10n) file. For simplicity, I'll include a few common phrases, but you can expand this as needed:

Create a new file **en-AU/translation.json:**

```
// locales/en-AU/translation.json
{
  "welcome": "G'day! Welcome to My App",
  "intro": "This is a ripper of a localized component, mate.",
}
```

In this example:

"G'day" is a friendly Australian greeting.

"Ripper" is an Australian slang term for excellent or fantastic.

"Mate" is a common term of address in Australian English.

Modify the i18n initialization in App.js to include 'en-AU':

```
// App.js

import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-
router-dom';
import { initReactI18next } from 'react-i18next';
import i18n from 'i18next';
import LanguageDetector from 'i18next-browser-languagedetector';
import LocalizedComponent from './components/LocalizedComponent';

import enTranslation from './locales/en/translation.json'; //
Default English
import frTranslation from './locales/fr/translation.json'; //
French
import enAUTranslation from './locales/en-AU/
translation.json'; // Australian English
```

```
i18n.use(initReactI18next)
  .use(LanguageDetector)
  .init({
    resources: {
      en: { translation: enTranslation },
      fr: { translation: frTranslation },
      'en-AU': { translation: enAUTranslation }, // Australian
English
    },
    fallbackLng: 'en',
    interpolation: {
      escapeValue: false,
    },
  });

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<LocalizedComponent />} />
      </Routes>
    </Router>
  );
};
export default App;
```

Now, your application can be localized for Australian English by including the 'en-AU' language resource. Adjust the content in translation.json according to the specific linguistic and cultural nuances you want to incorporate for the Australian audience.

You can run a Google Search for query ‘How to set the default language in my browser’. Set it to Australian English Then, you can test whether your app shows the content in Australian English.

Please note that if you're testing language-based features or internationalization (i18n) in your web application, you may also need to adjust the accept-lan-

guage headers in your browser's developer tools for accurate testing. You can learn about it here. Refer: https://localizely.com/blog/accept-language-header/

Chapter 21: Build and Deployment

Building and deploying a React application involves several steps to prepare your project for production and make it accessible to users. Here's a general guide on how to build and deploy a React application:

Step 1: Create a Production Build

Before deploying your React application, you need to create a production-ready build. You can do this using the following command in your project directory:

```
npm run build
```

This command generates an optimized, minified, and bundled version of your application in the build directory. This is what you'll deploy to your web server or hosting service.

Step 2: Choose a Hosting Service

You need a place to host your application files. Several hosting services are suitable for hosting React applications, including:

Netlify: Offers a simple and free hosting solution with continuous deployment from your Git repository.

Vercel: Provides a platform for hosting and deploying web applications with easy integration with Git repositories.

GitHub Pages: Allows you to host static sites directly from your GitHub repository.

AWS Amplify: Provides a serverless deployment platform with features like authentication and CI/CD.

Firebase Hosting: Part of the Firebase platform, it offers hosting for web apps, including React apps.

Choose a hosting service that suits your needs and set up an account or project with them.

Step 3: Configure Hosting

Depending on your chosen hosting service, you'll need to configure hosting settings. This typically involves connecting your hosting service to your project's repository (e.g., GitHub, GitLab) and specifying the build directory (usually build) in your project.

For example, if you're using Netlify, you can set up continuous deployment by linking your project's repository and specifying the build settings.

Step 4: Deploy Your Application

After configuring hosting, you can deploy your application. With most hosting services, this can be automated through continuous integration and continuous deployment (CI/CD) pipelines. Whenever you push changes to your repository, your hosting service will automatically build and deploy the new version.

For example, if you're using Netlify, every push to your repository's main branch will trigger a new deployment.

Step 5: Configure Domain and SSL

If you have a custom domain for your application, configure it to point to your hosting service. Most hosting services offer guides on how to set up custom domains.

Additionally, enable SSL (HTTPS) to secure your application. Most hosting services provide free SSL certificates or allow you to easily set up your own.

Step 6: Test Your Deployed Application

After deployment, thoroughly test your application on the production server to ensure everything works as expected. Check for any issues that might have arisen during the deployment process.

Step 7: Monitor and Maintain

Once your React application is live, it's essential to monitor its performance, fix any issues that arise, and keep it up to date with the latest dependencies and security patches.

Step 8: Scale and Optimize (if necessary)

If your application experiences increased traffic, you may need to scale your hosting resources or optimize your application's performance. Hosting services typically provide scalability options to handle increased load.

Remember that the steps mentioned here are general guidelines. The specific process may vary depending on your chosen hosting service and your project's requirements. Always refer to the documentation and guides provided by your hosting service for detailed instructions.

Chapter 22: Performance Optimisation Tips in React JS App

Performance optimization in a React application is crucial for providing a smooth user experience. Here are some tips and best practices to improve the performance of your React application:

1. Use Production Builds: Always build your React application for production using tools like Webpack to minify and optimize your code.

```
npm run build
```

2. Code Splitting: Implement code splitting to load only the necessary code for each route or component. This reduces the initial load time and improves user experience.
3. Lazy Loading: Use React.lazy() and Suspense to lazy load components, especially those that are not immediately needed when the app loads.
4. Optimize Images: Compress and optimize images to reduce their size. You can use tools like ImageOptim, TinyPNG, or image compression plugins for your build tools.
5. Minimize and Compress CSS and JS: Minify and compress your CSS and JavaScript files to reduce file size. Use CSS-in-JS libraries that generate optimized CSS.

6. Tree Shaking: Ensure your JavaScript bundler (e.g., Webpack) is configured to perform tree shaking. This eliminates unused code from the final bundle.
7. React.memo: Use React.memo to memoize functional components to prevent unnecessary re-renders.
8. Memoization: Implement memoization techniques like caching with libraries like reselect for optimizing selectors in Redux.
9. Reduce Component Re-renders: Avoid unnecessary component re-renders by using React hooks like useMemo and useCallback.
10. Optimize Context Providers: Minimize the use of context providers, as they can lead to unnecessary re-renders if not used properly. Prefer using context for high-level state that doesn't change frequently.
11. Web Workers: Offload heavy computational tasks to web workers to prevent blocking the main thread.
12. Virtualization: Use virtualization libraries like react-virtualized or react-window for efficiently rendering large lists or tables.
13. Bundle Analysis: Use tools like Webpack Bundle Analyzer to visualize and optimize your bundle size.
14. Progressive Web App (PWA): Convert your app into a PWA to allow for offline access, faster loading, and caching of assets.
15. Service Workers: Implement service workers for caching assets and enabling background synchronization in PWAs.
16. HTTP/2: Use HTTP/2 to take advantage of multiplexing and reduced latency for loading assets.
17. Use Modern JavaScript Features: Write modern JavaScript code, use ES6 features, and keep your dependencies up to date.
18. Minimize Redux Usage: Use Redux sparingly and avoid using it for small, simple states that can be managed with local component state.

19. Remove Unused Dependencies: Regularly review and remove unused dependencies to keep your application lightweight.
20. Debounce and Throttle: Implement debounce and throttle techniques for handling events that can trigger multiple rapid re-renders.
21. Server-Side Caching: Implement server-side caching strategies to reduce the load on your server and improve response times.
22. Avoid Large Render Methods: Keep your render methods small and avoid complex logic within them.
23. Profiler: Use the React Profiler tool to identify performance bottlenecks in your application.
24. Measure and Optimize: Continuously monitor and measure the performance of your application using browser developer tools and third-party performance monitoring tools. Address performance issues as they arise.
25. Server-Side Rendering (SSR): Implement SSR using libraries like Next.js. SSR can improve perceived performance and SEO.

Remember that optimization should be done selectively, focusing on the parts of your application that have the most impact on user experience. Profiling and measuring performance regularly will help you identify areas that need improvement.

Bonus Chapter 23: Styling in React with Styled Components

Styled Components is a popular library for styling React components in a way that keeps your styles encapsulated and more maintainable. It allows you to write CSS in JavaScript, making your styles more predictable and easy to manage. Here's how to use Styled Components in a React application:

Step 1: Installation

First, you need to install Styled Components in your React project. You can do this using npm or yarn:

```
npm install styled-components
```

Step 2: Creating Styled Components

Styled Components work by defining styles in JavaScript and then attaching them to your components. You can create a styled component using the styled object:

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

const Button = styled.button`
  background-color: #0074d9;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  &:hover {
    background-color: #0056b3;
  }
```

```
`;

export default Button;
```

In this example, we've created a Button component with custom styles. You can then use this Button component just like any other React component:

```
import React from 'react';
import Button from './Button';

function App() {
  return (
    <div>
      <h1>Hello Styled Components</h1>
      <Button>Click Me</Button>
    </div>
  );
}

export default App;
```

Step 3: Dynamic Styling

Styled Components allow for dynamic styling based on component props. You can pass props to the styles and conditionally apply them. For example, you can change the button's background color based on a prop:

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

const Button = styled.button`
  background-color: ${(props) => (props.primary ? '#0074d9' :
'white')};
  color: ${(props) => (props.primary ? 'white' : '#333')};
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  &:hover {
    background-color: ${(props) => (props.primary ? '#0056b3' :
'#f1f1f1')};
  }
```

```
`;

export default Button;
```

You can use this dynamic button by passing the primary prop:

```
<Button primary>Primary Button</Button>
<Button>Secondary Button</Button>
```

Step 4: Global Styles

Styled Components also support global styles using the createGlobalStyle function:

```
import { createGlobalStyle } from 'styled-components';

const GlobalStyles = createGlobalStyle`
  body {
    font-family: Arial, sans-serif;
    background-color: #f7f7f7;
  }
`;

export default GlobalStyles;
```

Include this global style in your app's entry point:

```
import React from 'react';
import GlobalStyles from './GlobalStyles';
import App from './App';

function Main() {
  return (
    <>
      <GlobalStyles />
      <App />
    </>
  );
}
```

```
export default Main;
```

Styled Components make it easy to create and manage styles in your React application, and they provide a flexible way to handle dynamic and conditional styling based on component props. Additionally, the styles are encapsulated and don't leak into other components, making your code more maintainable.