React

Outline

1. **Introduction to React**
    - What is React?
    - History and Evolution
    - Benefits of Using React
2. **Setting Up Your Development Environment**
    - Installing Node.js and npm
    - Setting Up a React Project with Create React App
    - Overview of the Project Structure
3. **JSX: The Syntax Extension**
    - Understanding JSX
    - Embedding Expressions in JSX
    - JSX Best Practices
4. **Components in React**
    - Functional vs. Class Components
    - Creating Your First Component
    - Component Composition
5. **State and Props**
    - Understanding State and Props
    - Managing State in Class and Functional Components
    - Passing Props to Components
6. **Event Handling in React**
    - Handling Events

- Synthetic Events
- Event Binding

7. **Lifecycle Methods**
    - Overview of Lifecycle Methods
    - Commonly Used Lifecycle Methods
    - Using Lifecycle Methods in Class Components
8. **Hooks: The New Way to Handle State and Side Effects**
    - Introduction to Hooks
    - Using the State Hook (useState)
    - Using the Effect Hook (useEffect)
    - Custom Hooks
9. **Handling Forms in React**
    - Controlled vs. Uncontrolled Components
    - Handling Form Submissions
    - Validating Form Input
10. **React Router: Navigation in React**
    - Setting Up React Router
    - Creating Routes and Links
    - Route Parameters and Nested Routes
11. **State Management with Redux**
    - Introduction to Redux
    - Setting Up Redux in a React Project
    - Actions, Reducers, and Store
    - Connecting Redux with React Components
12. **Styling in React**
    - CSS-in-JS
    - Styled Components
    - CSS Modules
13. **Fetching Data**
    - Using Fetch API

    - Using Axios
    - Handling Asynchronous Operations
14. **Testing in React**
    - Introduction to Testing
    - Testing Components with Jest and React Testing Library
    - Snapshot Testing
15. **Deploying React Applications**
    - Building for Production
    - Deploying to GitHub Pages
    - Deploying to Vercel or Netlify
16. **Advanced Topics**
    - Context API
    - React Performance Optimization
    - Server-Side Rendering (SSR) with Next.js
17. **Conclusion**
    - Summary of Key Points
    - Further Learning Resources
    - Final Thoughts

Daily Plan

Day 1: Introduction and Setup

- Write the introduction chapter.
- Describe the process of setting up the development environment.
- Provide a walkthrough of creating a new React project.

Day 2: JSX and Components

- Explain JSX syntax and its best practices.
- Write about functional and class components.
- Show examples of creating and composing components.

Day 3: State, Props, and Event Handling

- Write about state and props.
- Explain event handling in React with examples.

Day 4: Lifecycle Methods and Hooks

- Cover lifecycle methods in class components.
- Introduce hooks and explain `useState` and `useEffect`.
- Provide examples of custom hooks.

Day 5: Forms, Routing, and Redux

- Write about handling forms in React.
- Cover React Router for navigation.
- Introduce Redux and explain how to set it up in a React project.

Day 6: Styling, Data Fetching, and Testing

- Write about different styling approaches in React.
- Explain how to fetch data using Fetch API and Axios.
- Cover testing components with Jest and React Testing Library.

Day 7: Deployment and Advanced Topics

- Write about deploying React applications.
- Cover advanced topics like Context API, performance optimization, and SSR with Next.js.
- Conclude the eBook with a summary and further resources.

Final Steps

- Review and edit the content for clarity and accuracy.
- Format the eBook and add images/screenshots where necessary.
- Compile the eBook into a PDF or other desired formats.

By following this plan, you should be able to create a comprehensive React developer eBook in one week. Good luck!

Sure! I'll provide a detailed explanation with code examples for each topic. Let's start with the first few topics. If you need further topics, just let me know, and I will continue from where we left off.

1. Introduction to React

What is React?
React is a JavaScript library for building user interfaces. It was developed by Facebook and is used for creating single-page applications. React allows developers to build web applications that can update and render efficiently in response to data changes.

History and Evolution
React was first released in 2013. It introduced the concept of a virtual DOM, which improved performance by minimizing direct manipulations of the DOM. Over time, React has evolved with new features like hooks and concurrent mode.

Benefits of Using React

- **Component-Based**: Build encapsulated components that manage their own state.
- **Declarative**: React makes it easy to create interactive UIs with its declarative approach.
- **Learn Once, Write Anywhere**: React can be used to develop applications for web, mobile, and desktop.

2. Setting Up Your Development Environment

Installing Node.js and npm

Node.js is a JavaScript runtime, and npm is the package manager for Node.js.

```
# Download and install Node.js from https://nodejs.org/
# Verify installation
node -v
npm -v
```

Setting Up a React Project with Create React App

Create React App is a tool to set up a modern web app by running one command.

```
# Install Create React App globally
npm install -g create-react-app

# Create a new React project
create-react-app my-react-app

# Navigate into the project directory
cd my-react-app

# Start the development server
npm start
```

Overview of the Project Structure

After setting up the project, you will see a directory structure like this:

```
my-react-app/
├── node_modules/
├── public/
├── src/
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   └── logo.svg
├── .gitignore
├── package.json
├── README.md
└── yarn.lock
```

3. JSX: The Syntax Extension

Understanding JSX

JSX is a syntax extension for JavaScript that looks similar to XML or HTML. It's used with React to describe what the UI should look like.

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

// This is equivalent to:
const element = React.createElement("h1", null, "Hello, world!");
```

Embedding Expressions in JSX

You can embed any JavaScript expression in JSX by wrapping it in curly braces `{}`.

```
const name = "John";
const element = <h1>Hello, {name}!</h1>;
```

JSX Best Practices

- Always close tags.
- Use camelCase for attribute names (e.g., `className` instead of `class`).
- Wrap multiple lines of JSX in a single parent element.

4. Components in React

Functional vs. Class Components

Components let you split the UI into independent, reusable pieces.

Functional Component:

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

Class Component:

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

Creating Your First Component

```
import React from "react";

function App() {
    return (
        <div>
            <h1>My First Component</h1>
        </div>
    );
}

export default App;
```

Component Composition
Components can be composed to build complex UIs.

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

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

export default App;
```

5. State and Props

Understanding State and Props

- **Props** (short for properties) are read-only attributes used to pass data from parent to child components.
- **State** is a mutable object that stores a component's dynamic data and determines how the component renders and behaves.

Managing State in Class and Functional Components
Class Component:

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

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

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

Functional Component with useState:

```
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>
    );
}
```

Passing Props to Components

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

function App() {
    return (
        <div>
            <Greeting name="Alice" />
            <Greeting name="Bob" />
        </div>
    );
}

export default App;
```

6. Event Handling in React

Handling Events

React events are named using camelCase, and you pass a function as the event handler.

```
function App() {
    function handleClick() {
        alert("Button clicked!");
    }

    return <button onClick={handleClick}>Click me</button>;
}

export default App;
```

Synthetic Events

React creates a synthetic event wrapper to ensure the events work identically across all browsers.

Event Binding

When using class components, you often need to bind `this` to the event handler.

```
class App extends React.Component {
    constructor(props) {
        super(props);
        this.handleClick = this.handleClick.bind(this);
    }

    handleClick() {
```

```
        alert("Button clicked!");
    }

    render() {
        return <button onClick={this.handleClick}>Click me</button>;
    }
}
```

7. Lifecycle Methods

Overview of Lifecycle Methods

React components go through a lifecycle of mounting, updating, and unmounting.

Commonly Used Lifecycle Methods

- `componentDidMount`: Called after the component is rendered.
- `componentDidUpdate`: Called after the component is updated.
- `componentWillUnmount`: Called before the component is unmounted.

Using Lifecycle Methods in Class Components

```
class App extends React.Component {
    componentDidMount() {
        console.log("Component did mount");
    }

    componentDidUpdate() {
        console.log("Component did update");
    }

    componentWillUnmount() {
        console.log("Component will unmount");
    }

    render() {
        return <div>Hello, world!</div>;
    }
}
```

8. Hooks: The New Way to Handle State and Side Effects

Introduction to Hooks

Hooks let you use state and other React features without writing a class. They were introduced in React 16.8.

Using the State Hook (useState)

```
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>
    );
}

export default Counter;
```

Using the Effect Hook (useEffect)

The `useEffect` hook lets you perform side effects in functional components.

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

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

    useEffect(() => {
        const timer = setInterval(() => {
            setCount((prevCount) => prevCount + 1);
        }, 1000);

        // Cleanup the interval on component unmount
        return () => clearInterval(timer);
    }, []);

    return <div>Count: {count}</div>;
}

export default Timer;
```

Custom Hooks

You can create your own hooks to reuse stateful logic between components.

```
import { useState, useEffect } from "react";

function useFetch(url) {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(true);

    useEffect(() => {
        fetch(url)
            .then((response) => response.json())
            .then((data) => {
                setData(data);
                setLoading(false);
            });
    }, [url]);

    return { data, loading };
}

export default useFetch;
```

9. Handling Forms in React

Controlled vs. Uncontrolled Components

Controlled components have their form data controlled by React state, while uncontrolled components store their data in the DOM.

Controlled Component:

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

function Form() {
    const [value, setValue] = useState("");

    const handleChange = (e) => {
        setValue(e.target.value);
    };

    const handleSubmit = (e) => {
        e.preventDefault();
        alert(`Submitted value: ${value}`);
    };

    return (
        <form onSubmit={handleSubmit}>
            <label>
                Name:
                <input type="text" value={value} onChange={handleChange} />
            </label>
            <button type="submit">Submit</button>
        </form>
    );
}

export default Form;
```

Handling Form Submissions

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

function Form() {
    const [value, setValue] = useState("");

    const handleChange = (e) => {
        setValue(e.target.value);
    };

    const handleSubmit = (e) => {
        e.preventDefault();
        alert(`Submitted value: ${value}`);
    };

    return (
        <form onSubmit={handleSubmit}>
            <label>
                Name:
                <input type="text" value={value} onChange={handleChange} />
```

```
            </label>
            <button type="submit">Submit</button>
        </form>
    );
}

export default Form;
```

Validating Form Input

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

function Form() {
    const [value, setValue] = useState("");
    const [error, setError] = useState("");

    const handleChange = (e) => {
        setValue(e.target.value);
    };

    const handleSubmit = (e) => {
        e.preventDefault();
        if (value === "") {
            setError("Name is required");
        } else {
            setError("");
            alert(`Submitted value: ${value}`);
        }
    };

    return (
        <form onSubmit={handleSubmit}>
            <label>
                Name:
                <input type="text" value={value} onChange={handleChange} />
            </label>
            {error && <p style={{ color: "red" }}>{error}</p>}
            <button type="submit">Submit</button>
        </form>
    );
}

export default Form;
```

10. React Router: Navigation in React

Setting Up React Router

First, install `react-router-dom`.

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

Creating Routes and Links

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

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

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

function App() {
    return (
        <Router>
            <div>
                <nav>
                    <ul>
                        <li>
                            <Link to="/">Home</Link>
                        </li>
                        <li>
                            <Link to="/about">About</Link>
                        </li>
                    </ul>
                </nav>
                <Switch>
                    <Route path="/about">
                        <About />
                    </Route>
                    <Route path="/">
                        <Home />
                    </Route>
                </Switch>
            </div>
        </Router>
    );
}

export default App;
```

Route Parameters and Nested Routes

```
import React from "react";
import {
    BrowserRouter as Router,
    Route,
    Link,
    Switch,
    useParams,
    useRouteMatch,
} from "react-router-dom";

function Topic() {
    let { topicId } = useParams();
    return <h3>Requested topic ID: {topicId}</h3>;
}

function Topics() {
```

```
    let { path, url } = useRouteMatch();
    return (
        <div>
            <h2>Topics</h2>
            <ul>
                <li>
                    <Link to={`${url}/components`}>Components</Link>
                </li>
                <li>
                    <Link to={`${url}/props-v-state`}>Props v. State</Link>
                </li>
            </ul>
            <Switch>
                <Route exact path={path}>
                    <h3>Please select a topic.</h3>
                </Route>
                <Route path={`${path}/:topicId`}>
                    <Topic />
                </Route>
            </Switch>
        </div>
    );
}

function App() {
    return (
        <Router>
            <div>
                <nav>
                    <ul>
                        <li>
                            <Link to="/">Home</Link>
                        </li>
                        <li>
                            <Link to="/topics">Topics</Link>
                        </li>
                    </ul>
                </nav>
                <Switch>
                    <Route path="/topics">
                        <Topics />
                    </Route>
                    <Route path="/">
                        <h2>Home</h2>
                    </Route>
                </Switch>
            </div>
        </Router>
    );
}

export default App;
```

Feel free to ask for more topics or further details on any specific topic!

Sure, let's continue with the next set of topics.

11. State Management with Redux

Introduction to Redux
Redux is a state management library for JavaScript applications, often used with React for managing application state in a more predictable and maintainable way.

Setting Up Redux in a React Project
First, install Redux and React-Redux.

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

Actions, Reducers, and Store

- **Actions**: Plain JavaScript objects that describe what happened.
- **Reducers**: Functions that take the current state and an action, and return a new state.
- **Store**: An object that holds the application state and provides methods to access state, dispatch actions, and subscribe to changes.

Example Setup

```
// actions.js
export const increment = () => ({
    type: "INCREMENT",
});

export const decrement = () => ({
    type: "DECREMENT",
});
```

```
// reducers.js
const initialState = {
    count: 0,
};

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

export default counterReducer;
```

```
// store.js
import { createStore } from "redux";
import counterReducer from "./reducers";

const store = createStore(counterReducer);
```

```
export default store;
```

Connecting Redux with React Components

```
// App.js
import React from "react";
import { useSelector, useDispatch } from "react-redux";
import { increment, decrement } from "./actions";

function App() {
    const count = useSelector((state) => state.count);
    const dispatch = useDispatch();

    return (
        <div>
            <h1>Count: {count}</h1>
            <button onClick={() => dispatch(increment())}>Increment</button>
            <button onClick={() => dispatch(decrement())}>Decrement</button>
        </div>
    );
}

export default App;
```

Provider Setup

```
// index.js
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import store from "./store";
import App from "./App";

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById("root")
);
```

12. Styling in React

CSS-in-JS

CSS-in-JS is a technique where JavaScript is used to style components.

Styled Components

Styled Components is a popular CSS-in-JS library.

```
npm install styled-components
```

```
// App.js
import React from "react";
import styled from "styled-components";

const Button = styled.button`
    background-color: #4caf50;
```

```
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;

    &:hover {
        background-color: #45a049;
    }
`;

function App() {
    return (
        <div>
            <Button>Styled Button</Button>
        </div>
    );
}

export default App;
```

CSS Modules

CSS Modules locally scope CSS by automatically generating unique class names.

```
/* App.module.css */
.button {
    background-color: #4caf50;
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}

.button:hover {
    background-color: #45a049;
}
```

```
// App.js
import React from "react";
import styles from "./App.module.css";

function App() {
    return (
        <div>
            <button className={styles.button}>CSS Module Button</button>
        </div>
    );
}

export default App;
```

13. Fetching Data

Using Fetch API

Fetch API is used to make network requests.

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

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

    useEffect(() => {
        fetch("https://api.example.com/data")
            .then((response) => response.json())
            .then((data) => setData(data));
    }, []);

    if (!data) {
        return <div>Loading...</div>;
    }

    return (
        <div>
            <h1>Data from API</h1>
            <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
    );
}

export default App;
```

Using Axios

Axios is a promise-based HTTP client for JavaScript.

```
npm install axios
```

```
import React, { useState, useEffect } from "react";
import axios from "axios";

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

    useEffect(() => {
        axios
            .get("https://api.example.com/data")
            .then((response) => setData(response.data));
    }, []);

    if (!data) {
        return <div>Loading...</div>;
    }

    return (
        <div>
            <h1>Data from API</h1>
            <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
    );
}

export default App;
```

Handling Asynchronous Operations
You can handle asynchronous operations using async/await syntax.

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

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

    useEffect(() => {
        const fetchData = async () => {
            const response = await fetch("https://api.example.com/data");
            const result = await response.json();
            setData(result);
        };

        fetchData();
    }, []);

    if (!data) {
        return <div>Loading...</div>;
    }

    return (
        <div>
            <h1>Data from API</h1>
            <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
    );
}

export default App;
```

14. Testing in React

Introduction to Testing
Testing is essential to ensure your code works as expected. In React, testing can be done using various libraries like Jest and React Testing Library.

Testing Components with Jest and React Testing Library
First, install the necessary packages.

```
npm install @testing-library/react @testing-library/jest-dom
```

Writing Tests
Create a test file for your component (e.g., `App.test.js`).

```
// App.js
import React from "react";

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

export default App;
```

```
// App.test.js
import React from "react";
import { render, screen } from "@testing-library/react";
import App from "./App";

test("renders hello world", () => {
    render(<App />);
    const element = screen.getByText(/hello, world/i);
    expect(element).toBeInTheDocument();
});
```

Snapshot Testing
Snapshot testing ensures that the UI does not change unexpectedly.

```
import React from "react";
import renderer from "react-test-renderer";
import App from "./App";

test("renders correctly", () => {
    const tree = renderer.create(<App />).toJSON();
    expect(tree).toMatchSnapshot();
});
```

15. Deploying React Applications

Building for Production
Build your React application for production.

```
npm run build
```

This will create a `build` directory with your optimized application.

Deploying to GitHub Pages
Install the `gh-pages` package.

```
npm install gh-pages --save-dev
```

Add the following scripts to your `package.json`.

```
"homepage": "https://your-username.github.io/your-repo-name",
"scripts": {
  "predeploy": "npm run build",
  "deploy": "gh-pages -d build"
}
```

Deploy your application.

```
npm run deploy
```

Deploying to Vercel or Netlify
Vercel and Netlify are popular platforms for deploying React applications. Follow their documentation to deploy your app.

16. Advanced Topics

Context API

The Context API provides a way to pass data through the component tree without having to pass props down manually at every level.

```
import React, { createContext, useContext } from "react";

const MyContext = createContext();

function MyProvider({ children }) {
    const value = "Hello from context!";
    return <MyContext.Provider value={value}>{children}</MyContext.Provider>;
}

function MyComponent() {
    const value = useContext(MyContext);
    return <div>{value}</div>;
}

function App() {
    return (
        <MyProvider>
            <MyComponent />
        </MyProvider>
    );
}

export default App;
```

React Performance Optimization

Optimize performance using techniques like code splitting, memoization, and avoiding unnecessary re-renders.

Server-Side Rendering (SSR) with Next.js

Next.js is a React framework that enables server-side rendering and static site generation.

```
npx create-next-app my-next-app
```

Create a page.

```
// pages/index.js
import React from "react";

function HomePage() {
    return <div>Welcome to Next.js!</div>;
}

export default HomePage;
```

Run the development server.

```
npm run dev
```

This concludes the detailed explanation with code examples for key topics in React development. If you have specific questions or need further details on any topic, feel free to ask!

Certainly, let's continue with more advanced topics and best practices in React development.

17. Context API and Hooks

Using Context with Functional Components
The Context API can be combined with hooks to manage global state in a more functional way.

Creating and Using Context

```
import React, { createContext, useState, useContext } from "react";

const MyContext = createContext();

function MyProvider({ children }) {
    const [value, setValue] = useState("Hello from context!");
    return (
        <MyContext.Provider value={{ value, setValue }}>
            {children}
        </MyContext.Provider>
    );
}

function MyComponent() {
    const { value, setValue } = useContext(MyContext);
    return (
        <div>
            <p>{value}</p>
            <button onClick={() => setValue("New value from context!")}>
                Change Value
            </button>
        </div>
    );
}

function App() {
    return (
        <MyProvider>
            <MyComponent />
        </MyProvider>
    );
}

export default App;
```

18. Code Splitting and Lazy Loading

Code Splitting with React.lazy and Suspense
Code splitting helps in loading parts of the application on demand, which improves the performance.

```
import React, { Suspense, lazy } from "react";

const OtherComponent = lazy(() => import("./OtherComponent"));

function App() {
    return (
        <div>
```

```
            <Suspense fallback={<div>Loading...</div>}>
                <OtherComponent />
            </Suspense>
        </div>
    );
}

export default App;
```

19. Error Boundaries

Using Error Boundaries

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI.

```
import React from "react";

class ErrorBoundary extends React.Component {
    constructor(props) {
        super(props);
        this.state = { hasError: false };
    }

    static getDerivedStateFromError(error) {
        return { hasError: true };
    }

    componentDidCatch(error, info) {
        console.log(error, info);
    }

    render() {
        if (this.state.hasError) {
            return <h1>Something went wrong.</h1>;
        }

        return this.props.children;
    }
}

function BuggyComponent() {
    throw new Error("I crashed!");
    return <div>This will not be rendered.</div>;
}

function App() {
    return (
        <ErrorBoundary>
            <BuggyComponent />
        </ErrorBoundary>
    );
}

export default App;
```

20. Performance Optimization Techniques

Using React.memo for Performance Optimization

`React.memo` is a higher order component that memoizes the rendered output of a component to prevent unnecessary re-renders.

```
import React, { useState, memo } from "react";

const ChildComponent = memo(({ name }) => {
    console.log("ChildComponent render");
    return <div>Hello, {name}</div>;
});

function App() {
    const [count, setCount] = useState(0);
    const [name, setName] = useState("John");

    return (
        <div>
            <button onClick={() => setCount(count + 1)}>Increment</button>
            <ChildComponent name={name} />
        </div>
    );
}

export default App;
```

Using useMemo and useCallback Hooks

`useMemo` and `useCallback` hooks help in optimizing performance by memoizing values and functions.

```
import React, { useState, useMemo, useCallback } from "react";

function ExpensiveCalculation({ number }) {
    const compute = useMemo(() => {
        console.log("Computing...");
        return number * 2;
    }, [number]);

    return <div>{compute}</div>;
}

function App() {
    const [count, setCount] = useState(0);
    const [number, setNumber] = useState(1);

    const increment = useCallback(() => {
        setCount(count + 1);
    }, [count]);

    return (
        <div>
            <button onClick={increment}>Increment</button>
            <ExpensiveCalculation number={number} />
        </div>
    );
}
```

```
export default App;
```

21. Server-Side Rendering (SSR) with Next.js

Introduction to Next.js
Next.js is a popular React framework that provides SSR, static site generation, and many other features out of the box.

Setting Up a Next.js Project

```
npx create-next-app my-next-app
cd my-next-app
npm run dev
```

Creating Pages
Next.js uses a file-based routing system. Create files in the `pages` directory to create routes.

```
// pages/index.js
import React from "react";

function HomePage() {
    return <div>Welcome to Next.js!</div>;
}

export default HomePage;
```

SSR with getServerSideProps
Fetch data on each request.

```
// pages/index.js
import React from "react";

export async function getServerSideProps() {
    const res = await fetch("https://api.example.com/data");
    const data = await res.json();
    return { props: { data } };
}

function HomePage({ data }) {
    return (
        <div>
            <h1>Data from API</h1>
            <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
    );
}

export default HomePage;
```

22. Static Site Generation (SSG) with Next.js

SSG with getStaticProps
Generate static pages at build time.

```
// pages/index.js
import React from "react";

export async function getStaticProps() {
    const res = await fetch("https://api.example.com/data");
    const data = await res.json();
    return { props: { data } };
}

function HomePage({ data }) {
    return (
        <div>
            <h1>Data from API</h1>
            <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
    );
}

export default HomePage;
```

SSG with getStaticPaths
Generate static paths for dynamic routes.

```
// pages/posts/[id].js
import React from "react";

export async function getStaticPaths() {
    const res = await fetch("https://api.example.com/posts");
    const posts = await res.json();
    const paths = posts.map((post) => ({ params: { id: post.id.toString() } }));
    return { paths, fallback: false };
}

export async function getStaticProps({ params }) {
    const res = await fetch(`https://api.example.com/posts/${params.id}`);
    const post = await res.json();
    return { props: { post } };
}

function PostPage({ post }) {
    return (
        <div>
            <h1>{post.title}</h1>
            <p>{post.body}</p>
        </div>
    );
}

export default PostPage;
```

23. React Native: Building Mobile Apps

Introduction to React Native
React Native is a framework for building native mobile apps using React.

Setting Up React Native
Install the React Native CLI and create a new project.

```
npx react-native init MyNewProject
cd MyNewProject
npx react-native start
```

Creating Components in React Native
React Native components are similar to React components but have different primitives.

```
// App.js
import React from "react";
import { View, Text, StyleSheet } from "react-native";

function App() {
    return (
        <View style={styles.container}>
            <Text style={styles.text}>Hello, React Native!</Text>
        </View>
    );
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: "center",
        alignItems: "center",
    },
    text: {
        fontSize: 20,
    },
});

export default App;
```

Handling Navigation in React Native
Use React Navigation for routing and navigation.

```
npm install @react-navigation/native @react-navigation/stack

// App.js
import React from "react";
import { NavigationContainer } from "@react-navigation/native";
import { createStackNavigator } from "@react-navigation/stack";
import HomeScreen from "./HomeScreen";
import DetailsScreen from "./DetailsScreen";

const Stack = createStackNavigator();

function App() {
    return (
        <NavigationContainer>
            <Stack.Navigator initialRouteName="Home">
                <Stack.Screen name="Home" component={HomeScreen} />
                <Stack.Screen name="Details" component={DetailsScreen} />
            </Stack.Navigator>
```

```
        </NavigationContainer>
    );
}

export default App;
```

24. Managing Side Effects with Redux Saga

Introduction to Redux Saga

Redux Saga is a middleware library for handling side effects in Redux applications.

Setting Up Redux Saga

Install Redux Saga.

```
npm install redux-saga
```

Creating Sagas

Sagas are generator functions that manage side effects.

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

function* fetchData(action) {
  try {
    const data = yield call(() => fetch('https://api.example.com/data').then(res =>
res.json()));
    yield put({ type: 'FETCH_SUCCEEDED', data });
  } catch (e) {
    yield put({ type: 'FETCH_FAILED', message: e.message });
  }
}

function* mySaga() {
  yield takeEvery('FETCH_REQUESTED

', fetchData);
}

export default mySaga;
```

Connecting Sagas to the Store

```
// store.js
import createSagaMiddleware from "redux-saga";
import { createStore, applyMiddleware } from "redux";
import mySaga from "./sagas";
import reducer from "./reducers";

const sagaMiddleware = createSagaMiddleware();
const store = createStore(reducer, applyMiddleware(sagaMiddleware));
sagaMiddleware.run(mySaga);

export default store;
```

25. Conclusion and Next Steps

In this eBook, we have covered a wide range of topics in React development, from the basics of components and state management to advanced concepts like SSR, performance optimization, and mobile app development with React Native.

Next Steps

- **Explore Advanced Topics**: Dive deeper into performance optimization, custom hooks, and advanced patterns.
- **Build Real Projects**: Apply what you've learned by building real-world applications.
- **Stay Updated**: React and its ecosystem are constantly evolving. Stay updated with the latest features and best practices.

Resources

- React Documentation
- Redux Documentation
- Next.js Documentation
- React Native Documentation

Feel free to ask for more details or any other topic you want to explore!