Module name & code:SWDFA501:FRONTEND APP. DEVELOPMENT WITH REACT.JS

Learning outcome 1: Develop React.js application.

- **PREPARATION OF REACT JS ENVIRONMENT**

  ✓ Definition of key concepts

  - **ReactJS:** A JavaScript library for building user interfaces, developed by Facebook. It allows you to build single-page applications with a component-based architecture.
  - **Component:** A reusable piece of UI that can be composed to build complex user interfaces. Components can be either class-based or functional.
  - **JSX (JavaScript XML):** A syntax extension for JavaScript that looks similar to HTML. It allows you to write HTML elements and components in a JavaScript file, which React will then render.
  - **Props:** Short for properties, props are read-only attributes used to pass data from a parent component to a child component.
  - **State:** A built-in object that holds data or information about the component's current situation. Unlike props, state is mutable and can be changed within the component.
  - **Lifecycle Methods:** Special methods in class components that are called at different stages of a component's lifecycle (e.g., componentDidMount, componentWillUnmount). They are used for setup, updates, and teardown.
  - **Hooks:** Functions that let you use state and other React features without writing a class. Examples include useState, useEffect, and useContext.
  - **Virtual DOM:** A lightweight copy of the real DOM. React uses it to optimize updates by comparing the virtual DOM with the real DOM and only making necessary changes.
  - **React Router:** A library for handling routing in React applications. It allows you to manage navigation and rendering of different components based on the URL.
  - **Redux:** A state management library for JavaScript applications. It helps manage and centralize the application state and allows components to access and update the state consistently.

  ✓ **INTRODUCTION**

    - Uses of react

➢ **Single Page Applications (SPAs)**: React is widely used to build SPAs where content updates dynamically without requiring a full

page reload. This creates a more fluid and responsive user experience.

- **User Interfaces for Web Apps**: React allows developers to build complex and interactive user interfaces with reusable components, making it easier to manage and scale large applications.
- **Mobile Applications**: With React Native, a framework built on React, you can create native mobile apps for iOS and Android using the same principles and components as in web development.
- **Server-Side Rendering (SSR)**: React can be used with frameworks like Next.js to render components on the server side, improving performance and SEO by generating HTML on the server before sending it to the client.
- **Static Site Generation (SSG)**: Similar to SSR, React can be used to pre-render pages at build time using tools like Gatsby, which can improve performance and load times for static websites.
- **Progressive Web Apps (PWAs)**: React can help build PWAs, which offer a native app-like experience on the web with features like offline support and push notifications.
- **Component Libraries**: React's component-based architecture is ideal for creating reusable component libraries and design systems, which can be shared across different projects or teams

Features

- **Component-Based Architecture:** React allows you to build encapsulated components that manage their own state and then compose them to make complex UIs.
- **JSX (JavaScript XML):** JSX allows you to write HTML elements and components in JavaScript, making the code more readable and easier to debug.
- **Virtual DOM:** React uses a virtual DOM to optimize updates and rendering, which improves performance by minimizing direct manipulation of the actual DOM.
- **Unidirectional Data Flow:** Data in React flows from parent to child components, which simplifies the debugging and management of application state.
- **Lifecycle Methods:** React provides lifecycle methods that allow you to run code at specific points in a component's life (e.g., mounting, updating, unmounting).
- **Hooks:** React Hooks (like useState, useEffect, etc.) allow you to use state and other React features without writing a class.
- **Context API:** Helps manage global state and pass data through the component tree without having to pass props manually at every level.

- **React Router:** Provides navigation and routing capabilities for single-page applications, enabling seamless transitions between different views or pages.
- **Server-Side Rendering (SSR):** React can be used with frameworks like Next.js to enable server-side rendering, which can improve SEO and initial load performance.

✓ **Installation of Node.js and Node Package Manager (NPM)**

Node.js is a JavaScript runtime that allows you to run JavaScript code outside of a browser. **NPM** is the package manager for Node.js, used to install and manage libraries and tools.

1. **Download Node.js:**
   - Visit the https://nodejs.org/en
   - Download the recommended version for your operating system.
   - Follow the installation instructions for your OS.
2. **Verify Installation:**
   - Open a terminal or command prompt.
   - Run **node -v** to check the installed Node.js version.
   - Run **npm -v** to check the installed NPM version.

✓ **Creating a React Application**

With Node.js and NPM installed, you can use create-react-app to set up a new React project quickly.

1. **Open a terminal or command prompt.**
2. **Run the following command to create a new React app:**

   npx create-react-app **my-app**

   Replace my-app with your desired project name.

3. **Navigate to your project directory:**

   cd my-app

4. **Start the development server:**

   npm start

   This will open your new React app in the browser at http://localhost:3000.

✓ **Explore React Project Structure**

A typical React project created with create-react-app has the following structure:

- **node_modules/**: Contains all project dependencies.
- **public/**: Contains static files like index.html, icons, etc.
- **src/**: Contains the React application code.
  - **App.js**: Main component of the app.
  - **index.js**: Entry point for the React app.
- **.gitignore**: Specifies files and directories to be ignored by Git.
- **package.json**: Contains metadata about the project and its dependencies.
- **README.md**: A markdown file with project information.

✓ **Installation of Additional React Tools and Libraries**

React Developer Tools is a browser extension that helps you inspect and debug React components.

1. Install React Developer Tools:

For Chrome:

- **Open Chrome Browser**: Launch Google Chrome on your computer.
- **Visit Chrome Web Store**: Go to the Chrome Web Store.
- **Search for React Developer Tools**: In the search bar, type "React Developer Tools."
- **Install the Extension**: Click on the "React Developer Tools" extension in the search results and then click the "Add to Chrome" button. Confirm the installation if prompted.
- **Verify Installation**: Once installed, you should see the React DevTools icon in the Chrome toolbar.

For Firefox:

- **Open Firefox Browser**: Launch Mozilla Firefox on your computer.
- **Visit Firefox Add-ons**: Go to the https://addons.mozilla.org/en-US/firefox/
- **Search for React Developer Tools**: In the search bar, type "React Developer Tools."
- **Install the Extension**: Click on the "React Developer Tools" add-on and then click the "Add to Firefox" button. Confirm the installation if prompted.
- **Verify Installation**: Once installed, you should see the React DevTools icon in the Firefox toolbar.

2. **Use the Tools:**
   a. Open your React app in the browser.
   b. Click on the React Developer Tools icon in your browser's toolbar.
   c. Use the "Components" and "Profiler" tabs to inspect and analyze your React components and performance.

To install additional React libraries and tools, you typically use package managers like npm (Node Package Manager) or Yarn. Here's a guide on how to install and set up some commonly used libraries and tools:

REACT DEVELOPER LIBRARIES

1. Install Node.js and npm

First, ensure you have Node.js and npm installed. You can download and install them from **https://nodejs.org/en**

2. Create a React Project

If you haven't already created a React project, you can use Create React App:

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

3. Install React Libraries and Tools

common libraries and tools you might want to install:

React Router

For routing in your application:

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

Or with Yarn:

```
yarn add react-router-dom
```

Redux

For state management:

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

Or with Yarn:

```
yarn add redux react-redux
```

Axios

For making HTTP requests:

```
npm install axios
```

Or with Yarn:

yarn add axios

Styled-Components

For styling components:

npm install styled-components

Or with Yarn:

yarn add styled-components

Formik

For handling forms:

npm install formik

Or with Yarn:

yarn add formik

React Hook Form

For form handling with hooks:

npm install react-hook-form

Or with Yarn:

bash
Copy code
yarn add react-hook-form

React Query

For fetching, caching, and syncing server state:

npm install react-query

Or with Yarn:

yarn add react-query

- **Applying React basics**

React Components

React components are the building blocks of a React application. They can be classified into two main types:

1. **Class Components**:

    - **Definition**: A class component is a JavaScript class that extends React.Component.
    - **Usage**: It includes lifecycle methods and manages its own state.
    - **Example**:

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

  componentDidMount() {
    // Code to run after the component mounts
  }

  render() {
    return <div>{this.state.count}</div>;
  }
}
```

2. **Functional Components**:

    - **Definition**: A functional component is a JavaScript function that returns JSX.
    - **Usage**: It is simpler and can use hooks to manage state and side effects.
    - **Example**:

```
function MyComponent() {
  const [count, setCount] = React.useState(0);

  React.useEffect(() => {
    // Code to run after the component mounts or updates
  }, []);

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

JSX (JavaScript XML)

- **Definition**: JSX is a syntax extension that allows you to write HTML-like code within JavaScript.
- **Usage**: It makes the code more readable and easier to write.
- **Example**:

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

Props (Properties)

- **Definition**: Props are inputs to components that allow data to be passed from a parent component to a child component.
- **Usage**: They are used to configure components and make them reusable.
- **Example**:

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

<Greeting name="Alice" />
```

Lifecycle Methods (for Class Components)

Lifecycle methods are special methods that get called at different stages of a component's lifecycle. Here are a few important ones:

1. **componentDidMount**:
   - **Usage**: Invoked immediately after a component is mounted (i.e., inserted into the tree).
   - **Example**:

```
componentDidMount() {
  console.log('Component mounted');
}
```

2. **componentDidUpdate**:
   - **Usage**: Invoked immediately after updating occurs. This method is not called for the initial render.
   - **Example**:

```
componentDidUpdate(prevProps, prevState) {
  if (this.props.someValue !== prevProps.someValue) {
    console.log('Component updated');
  }
}
```

3. **componentWillUnmount**:
   - **Usage**: Invoked immediately before a component is unmounted and destroyed.
   - **Example**:

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

- **APPLYING UI NAVIGATION**

to applying UI navigation with React Router:

1. Installing React Router

To use React Router, you need to install it in your React application. You can do this using npm or yarn:

npm install react-router-dom

or

yarn add react-router-dom

2. Configuring Routes

After installing React Router, you need to set up routes in your application. This involves defining which components should be rendered based on the URL path.

Example:

```
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import Home from './Home';
import About from './About';
import NotFound from './NotFound';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}

export default App;
```

3. Basic React Navigation

To navigate between routes, use the Link component from React Router. This component renders an anchor (<a>) tag without reloading the page.

Example:

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

function Navigation() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
  );
}
```

4. Handling 404 Pages

To handle 404 pages (i.e., routes that don't match any defined path), you can use a wildcard route (*) and render a "Not Found" component.

Example (included in the route configuration):

```
<Route path="*" element={<NotFound />} />
```

5. Redirects

You can redirect users to a different route programmatically or based on certain conditions using the Navigate component.

Example of redirecting programmatically:

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

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

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

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

Example of redirecting from one route to another:

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

function OldPage() {
  return <Navigate to="/new-page" />;
}
```

6. URL Parameters

To pass parameters in the URL, use route parameters in your path and access them using the useParams hook.

Example:

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

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

  return <div>User ID: {userId}</div>;
}
```

Route configuration:

```
<Route path="/user/:userId" element={<User />} />
```

7. Nested Routing

React Router supports nested routing, allowing you to render routes within other routes. This is useful for creating complex UIs where components have their own sub-routes.

Example:

```
import { Route, Routes } from 'react-router-dom';
import Dashboard from './Dashboard';
import Profile from './Profile';

function App() {
  return (
    <Routes>
      <Route path="/dashboard" element={<Dashboard />}>
        <Route path="profile" element={<Profile />} />
      </Route>
    </Routes>
  );
}
```

In the Dashboard component, use Outlet to render nested routes:

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

function Dashboard() {
  return (
    <div>
      <h2>Dashboard</h2>
      <Outlet />
    </div>
```

```
  );
}
```

APPLYING REACT HOOKS

React Hooks allow you to use state, side effects, context, and more in functional components without writing class components.

Identifying Hooks

1. State Hooks (useState)

The useState hook allows you to add state to functional components.

Syntax:

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

Example:

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

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

2. Effect Hooks (useEffect)

The useEffect hook lets you perform side effects (such as data fetching, updating the DOM, or subscribing to events) in function components.

Syntax:

```
useEffect(() => {
  // Your effect code here
  return () => {
    // Cleanup function (optional)
  };
}, [dependencies]); // Dependencies array
```

Example:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]);  // This effect only runs when `count` changes
```

3. Context Hooks (useContext)

The useContext hook allows you to access the context provided by a parent component without manually passing props.

Syntax:

```
const value = useContext(MyContext);
```

Example:

```
const ThemeContext = React.createContext();

function ThemeButton() {
  const theme = useContext(ThemeContext);
  return <button style={{ background: theme.background }}>Click
Me</button>;
}
```

4. Ref Hooks (useRef)

The useRef hook allows you to create a mutable object that persists across renders, typically used to reference DOM elements or store values.

Syntax:

```
const refContainer = useRef(initialValue);
```

Example:

```
function TextInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
```

5. Callback Hooks (useCallback)

The useCallback hook returns a memoized version of a function, which is useful when passing callbacks to child components to prevent unnecessary re-renders.

Syntax:

```
const memoizedCallback = useCallback(() => {
```

```
  // Your callback logic here
}, [dependencies]);
```

Example:

javascript

Copy code

```
const handleClick = useCallback(() => {
  console.log("Button clicked");
}, []);  // The function will not be re-created on every render
```

Hook Selection and Combination

When developing a React application, you might need to combine hooks to achieve the desired behavior.

Example of Combining Hooks:

```
function ExampleComponent() {
  const [count, setCount] = useState(0);
  const [data, setData] = useState(null);

  useEffect(() => {
    fetchData().then(setData);
  }, []);

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

we combine useState for managing local state and useEffect for data fetching.

Optimizing Performance

React's rendering process can sometimes lead to performance bottlenecks. You can use the following techniques to improve performance:

1. Memoizing Expensive Calculations (useMemo)

The useMemo hook allows you to memoize the result of a computation and avoid recalculating it unnecessarily.

Example:

```
const expensiveCalculation = useMemo(() => {
  return computeExpensiveValue(data);
}, [data]);
```

2. Memoizing Components (React.memo)

React.memo prevents unnecessary re-renders by memoizing the component and only re-rendering it when props change.

Example:

```
const MyComponent = React.memo(function MyComponent({ value }) {
  return <div>{value}</div>;
});
```

3. Using useCallback for Event Handlers

As discussed earlier, useCallback prevents the re-creation of functions on every render.

Handling Complex State Logic

When managing complex state, you can use the useReducer hook instead of useState to manage state transitions more explicitly.

1. Reducer Hook (useReducer)

The useReducer hook is useful when your state logic involves multiple sub-values or when the next state depends on the previous state.

Syntax:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Example:

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });
```

```
  return (
    <div>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'increment'
})}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement'
})}>Decrement</button>
    </div>
  );
}
```

Managing Global State

For managing global state across your application, you can use state management tools such as the Context API, Redux, MobX, or Zustand.

1. Context API

The Context API is built into React and allows you to manage global state without prop drilling.

Example:

```
const UserContext = React.createContext();

function App() {
  const [user, setUser] = useState("John");

  return (
    <UserContext.Provider value={user}>
      <Toolbar />
    </UserContext.Provider>
  );
}

function Toolbar() {
  return (
    <div>
      <UserProfile />
    </div>
  );
}

function UserProfile() {
  const user = useContext(UserContext);
  return <div>User: {user}</div>;
}
```

2. Redux

Redux is a state management library that centralizes the entire state of the application.

Example:

```
// Action
const increment = () => ({ type: "INCREMENT" });

// Reducer
function counterReducer(state = 0, action) {
  switch (action.type) {
    case "INCREMENT":
      return state + 1;
    default:
      return state;
  }
}

// Store
const store = createStore(counterReducer);

// Usage in component
function Counter() {
  const count = useSelector((state) => state);
  const dispatch = useDispatch();

  return (
    <div>
      <p>{count}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
    </div>
  );
}
```

3. MobX

MobX is a state management library that uses observables to track changes in the state.

4. Zustand

Zustand is a lightweight state management library that provides a simple API for global state management.

Example:

```
const useStore = create((set) => ({
  count: 0,
  increment: () => set((state) => ({ count: state.count + 1 })),
}));
```

```
function Counter() {
  const { count, increment } = useStore();
  return (
    <div>
      <p>{count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

IMPLEMENTATION OF EVENT HANDLING IN REACT

Event handling in React is a crucial aspect of managing user interactions. React uses a mechanism called **Synthetic Events** to ensure consistent event behavior across different browsers.

Description of Events

In React, events are similar to HTML events but are named using camelCase rather than lowercase (e.g., onClick vs. onclick).

Example of Event Handler:

```
javascript
Copy code
function handleClick() {
  alert('Button clicked!');
}

function App() {
  return <button onClick={handleClick}>Click Me</button>;
}
```

Here, onClick is a React event handler that runs when the button is clicked.

Types of Events

React supports various types of events. Some of the most commonly used events are:

Mouse Events:

- onClick: Fired when an element is clicked.
- onDoubleClick: Fired when an element is double-clicked.
- onMouseEnter: Triggered when the mouse enters an element.
- onMouseLeave: Triggered when the mouse leaves an element.

Keyboard Events:

- onKeyDown: Fired when a key is pressed down.
- onKeyUp: Fired when a key is released.
- onKeyPress: Fired when a key is pressed.

Form Events:

- onSubmit: Fired when a form is submitted.
- onChange: Triggered when the value of an input element changes.
- onFocus: Triggered when an input field is focused.

Touch Events:

- onTouchStart: Fired when a touch point is placed on the screen.
- onTouchMove: Fired when a touch point is moved along the screen.
- onTouchEnd: Fired when a touch point is removed.

Focus Events:

- onFocus: Fired when an element receives focus.
- onBlur: Fired when an element loses focus.

Synthetic Events

React wraps native browser events in a **SyntheticEvent** object, which works identically across all browsers. This ensures consistency in event behavior and also provides React with a way to manage event delegation more efficiently.

Example of Synthetic Event:

```
function handleClick(event) {
  console.log(event); // SyntheticEvent
  console.log(event.nativeEvent); // Native browser event
}
```

Event Bubbling

Event bubbling occurs when an event is triggered on a child element, and then the same event is triggered on all of its parent elements up to the root of the DOM tree unless stopped.

Example:

```
function ParentComponent() {
  const handleParentClick = () => alert('Parent clicked!');
  const handleChildClick = (event) => {
    event.stopPropagation(); // Prevents bubbling
```

```
    alert('Child clicked!');
  };

  return (
    <div onClick={handleParentClick}>
      <button onClick={handleChildClick}>Click Me</button>
    </div>
  );
}
```

Clicking the button will only trigger the handleChildClick because event.stopPropagation() prevents the click from bubbling up to the parent.

Debouncing and Throttling Events

Debouncing and **Throttling** are performance optimization techniques used to limit the rate at which a function is called, especially for high-frequency events like scrolling or resizing.

Debouncing:

Debouncing delays the execution of a function until after a specified time has passed since the last event.

Example:

```
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => func.apply(this, args), delay);
  };
}

const handleScroll = debounce(() => {
  console.log("Scrolling...");
}, 300);

window.addEventListener("scroll", handleScroll);
```

Throttling:

Throttling ensures that a function is called at most once every specified time period, regardless of how many times the event is triggered.

Example:

```
function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
```

```
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => (inThrottle = false), limit);
    }
  };
}

const handleResize = throttle(() => {
  console.log("Resizing...");
}, 300);

window.addEventListener("resize", handleResize);
```

Using Controlled Components

A **controlled component** is an input element where the value is controlled by React state. In such components, every state change is handled via event handlers, and the state determines the current value of the input.

Example:

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

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

  return (
    <form>
      <input type="text" value={inputValue} onChange={handleChange} />
    </form>
  );
}
```

The inputValue state controls the input field, making it a controlled component.

Passing Arguments to Event Handlers

1. Arrow Function (in JSX)

You can pass additional arguments to event handlers by wrapping the function call in an arrow function.

Example:

```
function handleClick(id) {
  console.log("Clicked item", id);
}

function ItemList() {
  return <button onClick={() => handleClick(1)}>Click Me</button>;
}
```

2. Bind Method

Another approach to pass arguments to event handlers is using bind.

Example:

```
function handleClick(id) {
  console.log("Clicked item", id);
}

function ItemList() {
  return <button onClick={handleClick.bind(null, 1)}>Click Me</button>;
}
```

8. Use Custom Hooks for Event Listeners

You can encapsulate event handling logic using **custom hooks**, improving reusability and clarity in your code.

Example:

```
function useWindowResize() {
  useEffect(() => {
    const handleResize = () => console.log("Window resized");
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  }, []);
}

function App() {
  useWindowResize();
  return <div>Resize the window and check the console</div>;
}
```

Handling Events on Dynamic Lists

Handling events for dynamically rendered elements (e.g., lists) involves using unique identifiers (like id) to distinguish between items.

Example:

```
function ItemList({ items }) {
  const handleClick = (id) => {
    console.log("Clicked item:", id);
  };

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>
          <button onClick={() => handleClick(item.id)}>{item.name}</button>
        </li>
      ))}
    </ul>
  );
}
```

Clicking on any button will log the respective item.id due to the event handler capturing it through closure.

- **IMPLEMENTATION OF API INTEGRATION**

API integration in React, covering setup, organizing API calls, handling responses, and security:

1. Initial Setup and Planning

a. Describe API

An **API (Application Programming Interface)** is a set of protocols and tools that allows two applications to communicate with each other. In React, APIs are used to fetch or send data to a server. A common choice for REST APIs is JSON data transmitted via HTTP methods (GET, POST, PUT, DELETE).

b. Dependencies Installation (Axios)

To handle HTTP requests in React, libraries like **Axios** are commonly used due to their simplicity and automatic support for JSON and request/response transformations.

Install Axios:

npm install axios

Basic Axios Example:

```
import axios from 'axios';

axios.get('/api/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Error fetching data:', error);
  });
```

2. Organizing API Calls

a. Defining and Grouping API Calls

It's a good practice to separate API logic into its own module or service. This makes the API calls reusable and maintainable.

Example of Grouping API Calls:

```
// apiService.js
import axios from 'axios';

const apiClient = axios.create({
  baseURL: 'https://api.example.com',
  timeout: 1000,
});

export const fetchData = () => apiClient.get('/data');
export const postData = (payload) => apiClient.post('/data', payload);
```

b. Handling Data Fetching and Responses

In a React component, you can use **hooks** to handle data fetching and manage state for API responses.

Using useEffect for Fetching Data:

```
import { useEffect, useState } from 'react';
import { fetchData } from './apiService';

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

  useEffect(() => {
    fetchData()
      .then(response => setData(response.data))
      .catch(err => setError(err.message));
  }, []);
```

```
  return (
    <div>
      {error ? <p>{error}</p> : <ul>{data.map(item => <li
key={item.id}>{item.name}</li>)}</ul>}
    </div>
  );
}
```

c. Error Handling

Handle errors by using Axios's .catch() method or a try-catch block if using async/await.

Basic Error Handling:

```
axios.get('/api/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    if (error.response) {
      console.error('Server responded with status:', error.response.status);
    } else if (error.request) {
      console.error('No response received:', error.request);
    } else {
      console.error('Error setting up request:', error.message);
    }
  });
```

d. Asynchronous Handling and Concurrency

For multiple asynchronous API requests, Promise.all() can handle concurrent requests.

Using Promise.all:

```
useEffect(() => {
  Promise.all([axios.get('/api/data1'), axios.get('/api/data2')])
    .then(([res1, res2]) => {
      console.log('Data 1:', res1.data);
      console.log('Data 2:', res2.data);
    })
    .catch(error => console.error('Error fetching data:', error));
}, []);
```

Alternatively, you can use async/await for cleaner syntax:

Async/Await Example:

```
useEffect(() => {
  const fetchData = async () => {
```

```
    try {
      const res1 = await axios.get('/api/data1');
      const res2 = await axios.get('/api/data2');
      console.log('Data 1:', res1.data);
      console.log('Data 2:', res2.data);
    } catch (error) {
      console.error('Error fetching data:', error);
    }
  };
  fetchData();
}, []);
```

3. Performing API Security and Testing

a. API Security

Ensure API security by using **authentication** (e.g., tokens, OAuth) and **HTTPS** for encrypted communication.

- **Authorization Headers**: Use tokens for secure access. You can add the token to Axios globally or per request.

Adding Authorization Headers:

```
axios.defaults.headers.common['Authorization'] = 'Bearer ' + token;

axios.get('/api/secure-data')
  .then(response => console.log(response.data))
  .catch(error => console.error('Error fetching secure data:', error));
```

b. Testing API Calls

Test your API integration using mock APIs, tools like **Postman**, or integrating testing frameworks like **Jest** for unit and integration testing.

Example of Testing with Jest:

```
import axios from 'axios';
import { fetchData } from './apiService';

jest.mock('axios');

test('fetches data successfully', async () => {
  const data = { data: [{ id: 1, name: 'Item 1' }] };
  axios.get.mockResolvedValueOnce(data);

  const result = await fetchData();
  expect(result.data).toEqual(data.data);
});
```

Learning Outcome 2: Apply Tailwind CSS Framework.

Applying Tailwind Utility Classes in React.js

Integrating Tailwind CSS in React.js

1. **Install Tailwind CSS**:
   - To add Tailwind CSS to a React project, start by installing Tailwind and its dependencies:

     Bash

     ```
     npm install -D tailwindcss postcss autoprefixer
     npx tailwindcss init
     ```

2. **Configuring Tailwind CSS**:
   - After installation, configure Tailwind by editing the `tailwind.config.js` file:

     ```
     module.exports = {
       content: ['./src/**/*.{js,jsx,ts,tsx}'],
       theme: {
         extend: {},
       },
       plugins: [],
     };
     ```

   - Then import Tailwind into your CSS file, typically `src/index.css`:

     ```
     @tailwind base;
     @tailwind components;
     @tailwind utilities;
     ```

Using Utility-First Fundamentals

- Tailwind follows a utility-first approach, where you compose styles using utility classes directly in your markup.
- Example: Creating a button with padding, background color, and rounded corners using utilities.

```
<button className="bg-blue-500 text-white py-2 px-4 rounded">
  Click me
</button>
```

Handling Hover, Focus, and Other States

- Tailwind allows you to apply styles based on different states such as hover, focus, and active.
- Example: Adding hover and focus states to a button.

```
<button className="bg-blue-500 hover:bg-blue-700 focus:outline-none focus:ring-2 focus:ring-blue-500">
  Hover me
</button>
```

Animation and Transitions

- Tailwind makes it easy to apply animations and transitions with built-in utilities.
- Example: Adding a transition effect when hovering over a card.

```
<div className="transition transform hover:scale-105 duration-300 ease-in-out">
  <p>Hover to zoom</p>
</div>
```

Flexbox and Grid

- Tailwind offers comprehensive utilities for building layouts using **Flexbox** and **CSS Grid**.
- **Flexbox Example**:

```
<div className="flex justify-center items-center h-screen">
  <div className="bg-blue-500 text-white p-5">Centered Content</div>
</div>
```

- **Grid Example**:

```
<div className="grid grid-cols-3 gap-4">
  <div className="bg-blue-200 p-4">Item 1</div>
  <div className="bg-blue-200 p-4">Item 2</div>
  <div className="bg-blue-200 p-4">Item 3</div>
</div>
```

Reusing Styles

- Use **Tailwind's `@apply` directive** to reuse common styles by applying utility classes within custom CSS.
- Example: Defining a reusable button style in a CSS file.

```
.btn-primary {
  @apply bg-blue-500 text-white py-2 px-4 rounded;
}
```

- Now you can use this style in your React component:

```
<button className="btn-primary">Primary Button</button>
```

Adding Custom Styles

- Tailwind allows you to extend or customize your design system in the tailwind.config.js file.
- Example: Adding a custom color.

```
module.exports = {
  theme: {
    extend: {
      colors: {
        brand: '#3490dc',
      },
    },
  },
};
```

- Use the custom color in your components:

```
jsx
Copy code
<div className="bg-brand text-white p-4">Custom Brand Color</div>
```

Functions & Directives

- Tailwind includes several **functions and directives** that help customize and enhance your styles, such as @screen, @apply, and @layer.
- Example: Using the @screen directive for responsive breakpoints.

```
.text-responsive {
  @apply text-base;
  @screen md {
    @apply text-lg;
  }
  @screen lg {
    @apply text-xl;
  }
}
```

Applying Responsive Design Principles

Responsive design is essential for creating web applications that provide an optimal user experience across a variety of devices and screen sizes. Here's a breakdown of key principles and techniques for implementing responsive design:

1. Mobile-First Approach

- **Definition**: The mobile-first approach emphasizes designing for the smallest screens first and progressively enhancing the design for larger screens.
- **Benefits**:
  - Prioritizes essential features and content, leading to a better user experience on mobile devices.
  - Ensures performance optimization as mobile devices often have limited bandwidth and processing power.
- **Implementation**: Start with styles for mobile devices using default CSS, then add media queries to enhance styles for larger screens.

```
css
Copy code
/* Mobile styles */
.button {
  padding: 10px 20px;
  font-size: 16px;
}

/* Larger screens */
@media (min-width: 768px) {
  .button {
    padding: 15px 30px;
    font-size: 18px;
  }
}
```

2. Flexible Grid Layouts

- **Definition**: Use flexible grid layouts that adapt to different screen sizes by using relative units (like percentages) instead of fixed units (like pixels).
- **Benefits**:
  - Ensures that elements adjust in size and arrangement according to the screen dimensions.
- **Implementation**: Use CSS Grid or Flexbox for creating layouts that can easily adapt.

Css

```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  gap: 16px;
}
```

3. Responsive Images and Media

- **Definition**: Ensure that images and media scale appropriately to fit different screen sizes without losing quality.
- **Techniques**:
  - Use max-width: 100% to ensure images resize within their parent containers.
  - Utilize the srcset attribute to serve different image sizes based on screen resolution.

Html

```
<img src="image-small.jpg"
    srcset="image-small.jpg 600w, image-medium.jpg 1200w, image-large.jpg 1800w"
    sizes="(max-width: 600px) 100vw, (max-width: 1200px) 50vw, 33vw"
    alt="Responsive Image">
```

4. Media Queries and Breakpoints

- **Definition**: Media queries are CSS techniques that apply styles based on specific conditions, such as screen width, height, or orientation.
- **Implementation**: Define breakpoints to adjust styles at various screen widths.

css

```
/* Base styles for mobile */
.container {
  padding: 10px;
}

/* Tablet styles */
@media (min-width: 600px) {
  .container {
    padding: 20px;
  }
}

/* Desktop styles */
@media (min-width: 1024px) {
  .container {
    padding: 40px;
  }
}
```

5. Typography and Readability

- **Definition**: Ensure that text is legible and readable across all devices.
- **Techniques**:
  - Use relative units (like em or rem) for font sizes to ensure scalability.

- Maintain appropriate line height and contrast for readability.

Css

```
body {
  font-size: 16px; /* Base font size */
}

h1 {
  font-size: 2rem; /* 32px */
  line-height: 1.2;
}
```

6. Interactive Elements

- **Definition**: Design buttons, links, and other interactive elements to be easily tappable and accessible across devices.
- **Techniques**:
  - Ensure that touch targets are large enough (typically at least 44x44 pixels).
  - Provide visual feedback (like hover states) that works on both desktop and mobile.

Css

```
.button {
  padding: 10px 20px;
  border-radius: 5px;
  background-color: blue;
  color: white;
}

/* Hover and focus states */
.button:hover,
.button:focus {
  background-color: darkblue;
}
```

7. Testing and Iteration

- **Definition**: Continuously test the application on various devices and screen sizes to ensure responsiveness.
- **Techniques**:
  - Use browser developer tools to simulate different devices and screen sizes.
  - Gather user feedback to identify any issues or areas for improvement.
  - Iterate on the design based on testing results and user feedback to enhance usability and performance.

Customization of Tailwind Styles

Extending the Default Theme

- Tailwind allows you to extend the default theme by adding new values for colors, spacing, typography, and more.
- Modify the tailwind.config.js file to include custom extensions.

Example:

```
js
Copy code
module.exports = {
  theme: {
    extend: {
      spacing: {
        '128': '32rem',
        '144': '36rem',
      },
      colors: {
        primary: '#ff4757',
        secondary: '#3742fa',
      },
    },
  },
};
```

Adding Custom Variants

- Create custom variants to control when styles apply based on different states or conditions.
- Define these in the tailwind.config.js file.

Example:

Js

```
module.exports = {
  variants: {
    extend: {
      backgroundColor: ['active', 'group-hover'],
      textColor: ['visited'],
    },
  },
};
```

Custom Fonts and Typography

- To add custom fonts, first install the font using npm or link it in your HTML.
- Then extend the theme with the new font family.

Example:

Js

```
module.exports = {
  theme: {
    extend: {
      fontFamily: {
        sans: ['Inter', 'ui-sans-serif', 'system-ui'],
        serif: ['Merriweather', 'ui-serif', 'Georgia'],
      },
    },
  },
};
```

- Use the custom fonts in your components:

jsx

```
<h1 className="font-sans">Custom Font Heading</h1>
```

Customizing Colors

- Tailwind's color palette can be customized in the configuration file to include your brand colors or any specific shades.

Example:

js

```
module.exports = {
  theme: {
    extend: {
      colors: {
        brand: {
          light: '#4FD1C5',
          DEFAULT: '#38B2AC',
          dark: '#319795',
        },
      },
    },
  },
};
```

- Use the custom colors in your components:

Jsx

```
<div className="bg-brand text-white p-4">Brand Color Box</div>
```

Plugins for Additional Functionality

- Tailwind supports plugins to extend its functionality. You can use existing plugins or create your own.
- To use a plugin, install it and include it in your tailwind.config.js.

Example:

Bash

```
npm install @tailwindcss/forms
```

In your config:

Js

```
module.exports = {
  plugins: [
    require('@tailwindcss/forms'),
  ],
};
```

Custom Directives for Complex Designs

- Use Tailwind's directives like @apply for applying utility classes in your custom styles.
- This is particularly useful for creating reusable components or complex styles.

Example:

Css

```
.btn {
  @apply bg-blue-500 text-white font-bold py-2 px-4 rounded;
}
```

- You can then use the class in your components:

Jsx

```
<button className="btn">Button</button>
```

Conditional Styles with JavaScript

- For dynamic styling based on component state or props, you can conditionally apply Tailwind classes using JavaScript.

  Example:

  Jsx

```
const Button = ({ isActive }) => {
  return (
    <button className={`py-2 px-4 ${isActive ? 'bg-blue-500' : 'bg-gray-400'}`}>
      Click me
    </button>
  );
};
```

- This approach allows you to toggle styles based on user interactions or state changes effectively.

Learning outcome 3: Develop NextJS Application

Applying TypeScript Basics

Environment Setup

1. **Installing TypeScript**:
   - You can install TypeScript globally or as a development dependency in your project.
   - Global installation:

     ```
     npm install -g typescript
     ```

   - Local installation:

     ```
     npm install --save-dev typescript
     ```

2. **Configuring TypeScript**:
   - Create a `tsconfig.json` file to configure your TypeScript project. You can initialize it with:

     ```
     npx tsc --init
     ```

   - The `tsconfig.json` file allows you to specify compiler options, include/exclude files, and set the target ECMAScript version.

   Example `tsconfig.json`:

   Json

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

Implementing Interface of Variables

- TypeScript allows you to define custom types using interfaces, which can be used to describe the shape of objects and enforce type checking.

  Example:

typescript

```
interface User {
  id: number;
  name: string;
  email: string;
}

const user: User = {
  id: 1,
  name: "John Doe",
  email: "john@example.com"
};
```

Handling Functions in TypeScript

- You can define function parameters and return types for better type safety.

  Example:

  Typescript

```
function greet(user: User): string {
  return `Hello, ${user.name}`;
}

console.log(greet(user)); // Output: Hello, John Doe
```

- You can also use function overloads to handle different types of parameters.

  Example:

  typescript

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
  return a + b;
}

console.log(add(2, 3)); // Output: 5
console.log(add("Hello, ", "world!")); // Output: Hello, world!
```

Data Handling

1. **API Data Validation**:
   - You can validate the shape of data received from APIs by creating interfaces and using them when processing API responses.

   Example:

   typescript

```
async function fetchUser(userId: number): Promise<User> {
  const response = await fetch(`https://api.example.com/users/${userId}`);
  const data: User = await response.json();
  return data;
}
```

2. **Form Validation**:
   - TypeScript can help validate form inputs by defining types for the form data and ensuring that all fields are properly filled.

Example:

Typescript

```
interface FormData {
  username: string;
  password: string;
}

function validateForm(data: FormData): boolean {
  return data.username.length > 0 && data.password.length > 6;
}
```

3. **Error Handling and Exceptions**:
   - Use `try...catch` blocks to handle errors and exceptions gracefully in TypeScript.

Example:

Typescript

```
async function getUserData(userId: number): Promise<User | null> {
  try {
    const user = await fetchUser(userId);
    return user;
  } catch (error) {
    console.error("Error fetching user data:", error);
    return null;
  }
}
```

Setting Up a Next.js Project

Preparation of Environment

1. **Prerequisites**:
   - Ensure you have **Node.js** (version 12.22.0 or later) installed.
   - You can check your Node.js version with:

```
node -v
```

2. **Package Manager**:
    - Install **npm** (comes with Node.js) or use **Yarn** for package management.

Project Creation

1. **Create a New Next.js Project**:
    - You can create a new Next.js project using Create Next App:

```
npx create-next-app@latest my-next-app
# or using Yarn
yarn create next-app my-next-app
```

    - Navigate into your project directory:

```
cd my-next-app
```

2. **Run the Development Server**:
    - Start the Next.js development server to see your app in action:

```
npm run dev
# or
yarn dev
```

    - Open your browser and navigate to http://localhost:3000.

Initial Development

1. **Creating Pages and Components**:
    - Next.js uses a **file-based routing** system. Each file in the pages directory automatically becomes a route.
    - For example, to create an "About" page, create a file named about.js in the pages folder:

jsx

```
// pages/about.js
const About = () => {
  return <h1>About Us</h1>;
};

export default About;
```

2. **Creating Components**:
    - Create reusable components in a separate components folder. For example:

jsx

```
// components/Header.js
const Header = () => {
  return <header><h1>My Next.js App</h1></header>;
};
```

```
export default Header;
```

3. **Using Components in Pages**:
    - Import and use components in your pages:

jsx

```
// pages/index.js
import Header from '../components/Header';

const Home = () => {
  return (
    <div>
      <Header />
      <h2>Welcome to my Next.js app!</h2>
    </div>
  );
};

export default Home;
```

Implementing Search Engine Optimization (SEO)

- Next.js provides a built-in `Head` component for managing the `<head>` section of your pages, which is crucial for SEO.

  Example:

  Jsx

```
import Head from 'next/head';

const Home = () => {
  return (
    <>
      <Head>
        <title>Home | My Next.js App</title>
        <meta name="description" content="Welcome to my Next.js app!" />
      </Head>
      <h2>Welcome to my Next.js app!</h2>
    </>
  );
};

export default Home;
```

- Additionally, you can use tools like **Next SEO** for more advanced SEO features.

Styling

1. **CSS Modules**:

- Next.js supports CSS Modules for scoped styles. Create a CSS module file alongside your component:

css

```
/* styles/Header.module.css */
.header {
  background-color: #0070f3;
  color: white;
  padding: 10px;
}
```

Jsx

```
// components/Header.js
import styles from '../styles/Header.module.css';

const Header = () => {
  return <header className={styles.header}><h1>My Next.js App</h1></header>;
};

export default Header;
```

2. **Global Styles**:
   - You can add global styles by importing a CSS file in pages/_app.js:

Jsx

```
// pages/_app.js
import '../styles/globals.css';

const MyApp = ({ Component, pageProps }) => {
  return <Component {...pageProps} />;
};

export default MyApp;
```

Caching Strategies

- **Static Generation**: Use getStaticProps to fetch data at build time and serve pre-rendered pages, improving performance and SEO.

  Example:

  Jsx

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data },
  };
}
```

```
const Home = ({ data }) => {
  return <div>{data.title}</div>;
};

export default Home;
```

- **Incremental Static Regeneration (ISR)**: Use `revalidate` to update static pages at intervals.

  Example:

  Jsx

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data },
    revalidate: 10, // Regenerate the page every 10 seconds
  };
}
```

- **Client-Side Caching**: Use libraries like SWR or React Query for efficient data fetching and caching.

Implementing Rendering Techniques in Next.js

Static Site Generation (SSG)

- **Definition**: SSG allows pages to be pre-rendered at build time. This is optimal for content that does not change often, providing fast load times and great SEO benefits.
- **How to Implement**:

  javascript

```
// pages/index.js
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data }, // Will be passed to the page component as props
  };
}

const Home = ({ data }) => {
  return <div>{data.title}</div>;
};

export default Home;
```

Server-Side Rendering (SSR)

- **Definition**: SSR generates HTML on each request. This is useful for pages that require up-to-date data for every user.
- **How to Implement**:

javascript

```
// pages/posts/[id].js
export async function getServerSideProps(context) {
  const { id } = context.params;
  const res = await fetch(`https://api.example.com/posts/${id}`);
  const post = await res.json();

  return {
    props: { post }, // Will be passed to the page component as props
  };
}

const Post = ({ post }) => {
  return <h1>{post.title}</h1>;
};

export default Post;
```

Incremental Static Regeneration (ISR)

- **Definition**: ISR allows you to create or update static pages after the site has been built. You can specify a time interval for regeneration, enabling up-to-date content without a full rebuild.
- **How to Implement**:

Javascript

```
// pages/index.js
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data },
    revalidate: 10, // Regenerate the page every 10 seconds
  };
}

const Home = ({ data }) => {
  return <div>{data.title}</div>;
};

export default Home;
```

Client-Side Rendering (CSR)

- **Definition**: CSR renders pages in the browser, fetching data using JavaScript. This method is ideal for user-specific data and dynamic content.
- **How to Implement**:

Javascript

```
// pages/index.js
import { useEffect, useState } from 'react';

const Home = () => {
  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();
  }, []);

  return <div>{data ? data.title : 'Loading...'}</div>;
};

export default Home;
```

Implementing Routing in Next.js

Key Concepts

1. **File-System Based Routing**
   - Next.js uses a file-based routing system. Each `.js` file in the `pages` directory automatically becomes a route.
   - For example, `pages/about.js` corresponds to the `/about` route.
2. **Dynamic Routes**
   - Dynamic routes allow you to create paths with variable segments using square brackets. For example, `pages/posts/[id].js` captures routes like `/posts/1`, `/posts/2`, etc.

Example:

Javascript

```
// pages/posts/[id].js
export async function getServerSideProps(context) {
  const { id } = context.params;
  const res = await fetch(`https://api.example.com/posts/${id}`);
  const post = await res.json();

  return {
    props: { post },
  };
}

const Post = ({ post }) => {
  return <h1>{post.title}</h1>;
};

export default Post;
```

3. **Nested Routes**
    - You can create nested routes by organizing files in folders. For example, `pages/blog/index.js` will serve the `/blog` route, while `pages/blog/[slug].js` will serve `/blog/some-post`.
4. **Link Component**
    - Use the `Link` component from `next/link` for client-side navigation between routes without full page reloads.

Example:

Jsx

```
import Link from 'next/link';

const NavBar = () => {
  return (
    <nav>
      <Link href="/">
        <a>Home</a>
      </Link>
      <Link href="/about">
        <a>About</a>
      </Link>
    </nav>
  );
};

export default NavBar;
```

5. **Programmatic Navigation**
    - You can navigate programmatically using the `useRouter` hook from `next/router`.

Example:

```
javascript
Copy code
import { useRouter } from 'next/router';

const Home = () => {
  const router = useRouter();

  const goToAbout = () => {
    router.push('/about');
  };

  return <button onClick={goToAbout}>Go to About</button>;
};

export default Home;
```

6. **API Routes**
    - API routes are built-in to Next.js, allowing you to create serverless functions. They are defined in the `pages/api` directory.

Example:

javascript

```
// pages/api/hello.js
export default function handler(req, res) {
  res.status(200).json({ message: 'Hello, World!' });
}
```

7. **Catch-all Routes**
   - Use catch-all routes to match multiple segments in a path. You can create these by adding three dots in brackets. For example, `pages/posts/[...slug].js`.

Example:

javascript

```
// pages/posts/[...slug].js
const Post = ({ slug }) => {
  return <h1>Slug: {slug.join('/')}</h1>;
};

export async function getServerSideProps(context) {
  const { slug } = context.params;
  return { props: { slug } };
}

export default Post;
```

Linking Components

- Use the `Link` component to connect different parts of your application seamlessly.

Example:

Jsx

```
import Link from 'next/link';

const Home = () => {
  return (
    <div>
      <Link href="/about">
        <a>About</a>
      </Link>
    </div>
  );
};

export default Home;
```

Programmatic Navigation

- You can programmatically navigate to different routes based on events or conditions.

  Example:

  Javascript

```
import { useRouter } from 'next/router';

const MyComponent = () => {
  const router = useRouter();

  const handleNavigation = () => {
    router.push('/contact');
  };

  return <button onClick={handleNavigation}>Contact Us</button>;
};

export default MyComponent;
```

Dynamic Routes

- Dynamic routing allows for flexibility in handling multiple paths with varying parameters.

  Example:

  Javascript

```
// pages/products/[id].js
export async function getServerSideProps(context) {
  const { id } = context.params;
  const res = await fetch(`https://api.example.com/products/${id}`);
  const product = await res.json();

  return { props: { product } };
}

const Product = ({ product }) => {
  return <h1>{product.name}</h1>;
};

export default Product;
```

Query Parameters

- Access query parameters in your components using the useRouter hook.

  Example:

  javascript

```
import { useRouter } from 'next/router';
```

```
const MyComponent = () => {
  const router = useRouter();
  const { query } = router;

  return <div>Query parameter: {JSON.stringify(query)}</div>;
};

export default MyComponent;
```

Creation of API in Next.js

1. Define the API Endpoint

- To define an API endpoint in Next.js, create a file in the pages/api directory. Each file corresponds to an API route.

Example:

Javascript

```
// pages/api/products.js
export default function handler(req, res) {
  res.status(200).json({ message: 'Hello from the Products API!' });
}
```

This sets up an API endpoint at /api/products that responds with a JSON message.

2. Handling Request Types

- You can handle different HTTP request types (GET, POST, PUT, DELETE) within the same endpoint by checking req.method.

Example:

Javascript

```
// pages/api/products.js
export default function handler(req, res) {
  if (req.method === 'GET') {
    // Handle GET request
    res.status(200).json({ message: 'Fetching products...' });
  } else if (req.method === 'POST') {
    // Handle POST request
    const newProduct = req.body; // Access data from the request body
    res.status(201).json({ message: 'Product created', product: newProduct });
  } else {
    // Handle unsupported request types
    res.setHeader('Allow', ['GET', 'POST']);
```

```
    res.status(405).end(`Method ${req.method} Not Allowed`);
  }
}
```

3. Using Dynamic API Routes

- To create dynamic routes, use square brackets in your file name. This allows you to capture dynamic segments in the URL.

Example:

javascript

```
// pages/api/products/[id].js
export default function handler(req, res) {
  const { id } = req.query; // Extract the dynamic ID from the URL

  if (req.method === 'GET') {
    // Handle fetching a product by ID
    res.status(200).json({ message: `Fetching product with ID: ${id}` });
  } else {
    res.setHeader('Allow', ['GET']);
    res.status(405).end(`Method ${req.method} Not Allowed`);
  }
}
```

This creates an endpoint like `/api/products/1`, where `1` is a dynamic ID.

4. Testing Your API

- Test your API endpoints using tools like Postman, cURL, or directly in your front-end code.

Using Fetch API in Frontend:

Javascript

```
const fetchProducts = async () => {
  const response = await fetch('/api/products');
  const data = await response.json();
  console.log(data);
};

// Call the function to fetch data
fetchProducts();
```

Using Postman:

1. Open Postman.
2. Select the request type (GET, POST, etc.).
3. Enter your API endpoint (e.g., `http://localhost:3000/api/products`).
4. Click "Send" to see the response.

Securing the Application in Next.js

1. Performing Client-Side Security

- **Client-Side Rendering (CSR) Security**
  - Ensure that sensitive data is not exposed in client-side code. Use environment variables for configurations.
  - Validate and sanitize user inputs to prevent attacks like XSS (Cross-Site Scripting).
- **Cross-Origin Resource Sharing (CORS)**
  - Configure CORS to restrict which domains can access your API. This prevents unauthorized access.

Example:

Javascript

```
// pages/api/your-api.js
export default function handler(req, res) {
  res.setHeader('Access-Control-Allow-Origin', 'https://yourdomain.com');
  // other logic
}
```

- **Session Management**
  - Use secure cookies for session management. Implement cookie flags like `HttpOnly` and `Secure` to enhance security.

Example:

Javascript

```
// Setting a secure cookie in an API route
res.setHeader('Set-Cookie', 'sessionId=your_session_id; HttpOnly; Secure; Path=/; SameSite=Strict');
```

- **Third-Party Libraries (Auth0)**
  - Use trusted libraries like Auth0 for user authentication and authorization. These libraries provide built-in security features.

2. Performing Server-Side Security

- **HTTPS Enforcement**
  - Always use HTTPS to encrypt data in transit. You can use services like Vercel or implement HTTPS on your server.
- **Server-Side Rendering (SSR) Security**
  - Validate data fetched on the server side to ensure it is safe and secure before rendering it.
- **API Routes Security**
  - Protect your API routes using middleware for authentication and authorization checks.

Example:

Javascript

```
// pages/api/protected-route.js
const authenticate = (req, res, next) => {
  const token = req.headers.authorization;

  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }
  // Validate the token
  next();
};

export default function handler(req, res) {
  authenticate(req, res, () => {
    // Your API logic
    res.status(200).json({ message: 'Protected data' });
  });
}
```

- **Content Security Policy (CSP)**
  - Implement CSP headers to prevent XSS attacks. Specify which sources can be loaded.

Example:

Javascript

```
// In your custom server or middleware
res.setHeader("Content-Security-Policy", "default-src 'self'; script-src 'self' https://apis.yourdomain.com");
```

- **Authentication**
  - Implement robust authentication methods, such as JWT (JSON Web Tokens) or OAuth, to protect user sessions.

3. Performing General Security Measures

- **Input Validation**
  - Always validate and sanitize user inputs on both client-side and server-side to prevent injection attacks.
- **Error Handling**
  - Avoid exposing sensitive information in error messages. Log errors on the server but provide generic messages to users.
- **Regular Updates**
  - Keep dependencies up-to-date to mitigate vulnerabilities. Use tools like `npm audit` to check for vulnerabilities.
- **Security Headers**
  - Implement various security headers (e.g., X-Content-Type-Options, X-Frame-Options, X-XSS-Protection) to enhance security.

Example:

javascript

```
// In your custom server or middleware
res.setHeader('X-Content-Type-Options', 'nosniff');
res.setHeader('X-Frame-Options', 'DENY');
```

Learning outcome 4: Apply Progressive Web Application

Apply Progressive Web Application (PWA)

Maintain Responsiveness

Leverage Progressive Enhancement

- **Implementation:** Start with a basic HTML layout. For example, create a simple webpage that displays essential content. Then, use CSS and JavaScript to enhance the experience. This way, users on older browsers still have access to the core content.

Example:

html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>PWA Example</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>Welcome to My PWA</h1>
  <p>This is a basic page.</p>
  <script src="script.js" defer></script>
</body>
</html>
```

Prioritize Mobile-First Design

- **Implementation:** Design the layout for mobile devices first, then progressively enhance for larger screens using media queries.

Example:

css

```
/* styles.css */
body {
  font-size: 16px;
}

/* Tablet styles */
@media (min-width: 768px) {
  body {
    font-size: 18px;
  }
}

/* Desktop styles */
@media (min-width: 1024px) {
```

```
  body {
    font-size: 20px;
  }
}
```

Utilize Performance Optimization Techniques

- **Implementation:** Minimize file sizes and optimize images for faster loading times. Use tools like Webpack for bundling and minimizing assets.

Example:

javascript

```
// Webpack configuration example
module.exports = {
  mode: 'production',
  optimization: {
    minimize: true,
  },
  module: {
    rules: [
      {
        test: /\.(png|jpg|gif|svg)$/,
        use: [
          {
            loader: 'file-loader',
            options: {
              name: '[path][name].[ext]',
              outputPath: 'images/',
              publicPath: 'images/'
            }
          }
        ]
      }
    ]
  }
};
```

Configuring Web Application Manifest

Creating and Configuring the Manifest File

- **Implementation:** Create a manifest.json file to define the app's metadata.

Example:

json

```
{
  "name": "My PWA",
  "short_name": "PWA",
  "start_url": "/index.html",
  "display": "standalone",
```

```
  "background_color": "#ffffff",
  "theme_color": "#000000",
  "icons": [
    {
      "src": "/images/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ]
}
```

Referencing the Manifest in Your HTML

- **Implementation:** Link the manifest file in your HTML.

Example:

Html

```
<link rel="manifest" href="/manifest.json">
```

Testing and Validation

- **Implementation:** Use tools like Lighthouse to check if your manifest is configured correctly.

Implementation of Service Workers

Describe Service Workers

- **Implementation:** Service workers are scripts that run in the background, enabling features like caching and offline access.

Registration and Installation

- **Implementation:** Register a service worker in your main JavaScript file.

Example:

javascript

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
    .then((registration) => {
      console.log('Service Worker registered with scope:', registration.scope);
    })
    .catch((error) => {
      console.error('Service Worker registration failed:', error);
    });
}
```

Caching Strategy Implementation

- **Implementation:** Implement a caching strategy within the service worker to manage asset caching.

Example:

javascript
Copy code

```
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('my-cache').then((cache) => {
      return cache.addAll(['/index.html', '/styles.css', '/script.js']);
    })
  );
});
```

Updating Service Worker

- **Implementation:** Handle updates to the service worker to ensure users receive the latest version.

Example:

javascript

```
self.addEventListener('activate', (event) => {
  const cacheWhitelist = ['my-cache-v2']; // updated cache name
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```