React Tutorials - Revision

Written by : Divyansh Singh (rgndunes)

Contact : LinkedIn

Topics Covered

- Session 1
  - State
  - setState
  - Async behaviour of setState in class components
  - Async behaviour of setState in functional components
- Session 2
  - Conditional Rendering
  - Lists and Keys
  - What will happen if we use index as key in a list
  - Virtual DOM
  - Working Forms
  - Formik
- Session 3
  - Hooks
  - Lifecycle Methods
  - Difference between Hooks and Lifecycle Method
  - Different hooks in React
    - useState
    - useEffect
    - useContext
    - useReducer
    - useCallback
    - useMemo
    - useRef
  - Different Lifecycle Methods
    - construction
    - componentDidMount
    - shouldComponentUpdate
    - componentDidUpdate
    - componentWillUnmount
    - componentDidCatch

    - Why are functional components preferred over class components
    - Why are hooks preferred over lifecycle methods
- Session 4
    - What is an API
    - Working with API's
        - Fetch
        - Axios
        - Superagent
    - Best among fetch, axios and superagent and why ?
    - Canceling request in axios
    - What is react-query ?
- Session 5
    - What are Routes ?
    - What is react-router-dom ?
    - react-router-dom v5
        - Basic Routing
        - Nested Routing
        - Route Matchers
        - useHistory
        - useLocation
        - useParams
        - Redirect
    - react-router-dom v6
        - Routes
        - Nested Route
        - useRoutes
        - useSearchParams
        - useParams
        - Outlet
    - Higher Order Component (HOC)
- Session 6
    - Lifting State Up
    - State Management
    - Zustand
    - React Context API
    - React Context API vs Zustand
- Session 7
    - React Composition Model

- Session 8
  - Redux
  - Why is Redux used ?
  - How does it solve prop-drilling ?
  - How does it go well with react ?
  - What is a Redux Store ?
  - What are Redux Actions ?
  - What are Redux Reducers ?
  - How to dispatch a Redux Action ?
  - mapDispatchToProps
  - mapStateToProps
  - What is react-redux ?
  - How is Redux different react-redux ?
  - A simple TodoList redux application ?
- Session 9
  - Middleware
  - Redux Thunk
  - Combining Reducers
  - Redux hooks
    - useDispatch
    - useStore
  - react-redux hooks
    - useSelector
    - useDispatch
    - useStore
- Session 10
  - An e-commerce application

Session 1

State

In React, state refers to the data or variables that determine a component's behavior and render information to the user. There are two types of components in React: class components and functional components.

Class Components: In class components, state is managed within the component class. The state is defined in the constructor method and can be accessed and updated using the `this.state` object. The `setState` method is used to update the state and re-render the component. Here's an example:

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

  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }

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

In the above example, the `Counter` component is a class component that has a state with a single property `count` initialized to 0. The `handleClick` method is called when the button is clicked, and it uses `setState` to increment the count by 1. This causes the component to re-render with the updated count.

Functional Components: In functional components, state is managed using the `useState` hook. The `useState` hook allows functional components to have a state and it returns an array with two values, the first one is the current state and the second one is the function to update the state. Here's an example:

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

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

  function handleClick() {
    setCount(count + 1);
  }

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

setState

In React, `setState` is a method used to update the state of a component and re-render the component with the updated state. The behavior of `setState` is slightly different between class components and functional components.
Class Components: In class components, `setState` is a method that is available on the `this` object within the component class. It takes an object as an argument, which represents the new state, and it will merge the new state with the current state. Here is an example:

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

  handleClick() {
    this.setState({ count: this.state.count + 1 });
```

```
  }

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

In the above example, the `Counter` component has a state with a single property `count` initialized to 0. The `handleClick` method is called when the button is clicked, and it uses `setState` to increment the count by 1. This causes the component to re-render with the updated count.
Functional Components: In functional components, state is managed using the `useState` hook. The `useState` hook allows functional components to have a state and it returns an array with two values, the first one is the current state and the second one is the function to update the state.
The `useState` hook takes an initial value as an argument, it will use this value as the initial state and return the current state and the function to update the state, this function is similar to `setState` in class components.
Here's an example of how `useState` can be used in a functional component:

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

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

  function handleClick() {
    setCount(count + 1);
  }

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

In the above example, the `Counter` component is a functional component that uses the `useState` hook to manage the state. The `count` variable holds the

current count and the `setCount` function is used to update the count and re-render the component with the updated state.
When the `setCount` function is called, it will set the new state and re-render the component with the updated state, this causes the component to re-render with the updated count, the same behavior as `setState` in class components.
It's important to note that the `setCount` function will completely replace the current state with the new state, it will not merge the new state with the current state like `setState` method.

Async behaviour of setState in class components

In React, `setState` is an asynchronous operation, which means that state updates may not happen immediately, this behavior applies specifically to class components.

When a component calls `setState`, React will update the component's state and schedule a re-render. However, React will batch multiple state updates together to optimize performance. This means that if a component calls `setState` multiple times in a short period, React will wait until all the updates are finished before re-rendering the component.

This means that the component's state may not be updated immediately after a call to `setState`. For example, the following code will not work as expected:

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

  handleClick() {
    this.setState({ count: this.state.count + 1 });
    console.log(this.state.count); // still 0
  }

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

```
}
```

In this example, when the button is clicked, the `handleClick` method is called and it calls `setState` to increment the count by 1. However, the `console.log` statement will still log 0 because the state has not yet been updated.

This can create confusion, especially if you're trying to update the state based on the previous state. To avoid this problem, React provides a callback function as a second argument of the `setState` method, this callback will be invoked after the component's state has been updated:

```
this.setState(
  { count: this.state.count + 1 },
  () => console.log(this.state.count) // 1
);
```

This ensures that the callback will be executed after the state has been updated.

It's important to keep in mind that when you call `setState` multiple times in a short period, React will batch these updates together, so you should always use the callback function to ensure the execution order of your code.

Async behaviour of setState in functional components

In React, `setState` is an asynchronous operation that applies specifically to class components, as functional components use hooks such as `useState` for state management.

In functional components, the `useState` hook is used to manage state, it returns an array containing the current state and a function to update the state. The state updates are batched as well, in order to optimize performance.

However, unlike class components, functional components are re-rendered on every render cycle, this means that if you call the state update function multiple times within a render cycle, it will only pick up the last state update, discarding the previous ones. This behavior is known as the stale closure problem.

To avoid this problem, you can use the functional updates form of the state update function, which is a callback function that takes the previous state and returns the new state:

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

function handleClick() {
  setCount(prevCount => prevCount + 1);
}
```

This ensures that the callback will receive the most recent state and not the state that was in place when the callback was created.

It's important to keep in mind that when you call the state update function multiple times within the same render cycle, it will only pick up the last state update, discarding the previous ones.

Additionally, React also provides `useEffect` hook to handle side-effects such as network requests, timers, and subscriptions, this hook allows you to synchronize your component with an external system, it provides a way to synchronize your component with the effects of your state updates.

Session 2

Conditional Rendering

Conditional rendering in React is a technique that allows you to conditionally render elements or components based on certain conditions. This can be useful for displaying different content based on the user's actions, the state of the application, or any other condition.

There are several ways to implement conditional rendering in React:

1. Using the ternary operator: You can use the ternary operator to conditionally render an element based on a condition. Here is an example:

```
function Component({isLoading}) {
  return (
    <div>
      {isLoading ? <p>Loading...</p> : <p>Data Loaded</p>}
```

```
    </div>
  );
}
```

In this example, the component will render "Loading..." when the isLoading prop is true, and "Data Loaded" when it's false.

2. Using the if statement: You can also use the if statement to conditionally render an element based on a condition. Here is an example:

```
function Component({isLoading}) {
  let content;
  if (isLoading) {
    content = <p>Loading...</p>;
  } else {
    content = <p>Data Loaded</p>;
  }
  return <div>{content}</div>;
}
```

3. Using the && operator: A shorthand way of using the ternary operator, you can use the && operator to conditionally render an element based on a condition. Here is an example:

```
function Component({isLoading}) {
  return (
    <div>
      {isLoading && <p>Loading...</p>}
      {!isLoading && <p>Data Loaded</p>}
    </div>
  );
}
```

In this example, if the isLoading prop is true, the first element will be rendered, otherwise the second element will be rendered.

4. Using the switch statement: You can also use the switch statement to conditionally render based on multiple conditions. Here is an example:

```
function Component({status}) {
  let content;
  switch (status) {
    case 'loading':
      content = <p>Loading...</p>;
      break;
    case 'success':
      content = <p>Data Loaded</p>;
      break;
    case 'error':
```

```
      content = <p>Error</p>;
      break;
    default:
      content = null;
  }
  return <div>{content}</div>;
```

Lists and Keys

In React, lists are used to render multiple elements of the same type. Lists are often used to display data that is retrieved from an API or a database.

When rendering lists, React requires that each element in the list has a unique key. A key is a special attribute that React uses to identify which element has been added, removed, or changed. This allows React to efficiently update the list and minimize the number of DOM operations, which improves performance.

Here's an example of how to render a list of items using the `map` function:

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

In this example, the `items` prop is an array of objects, and the `List` component maps over the array and renders an `li` element for each item in the array. The `key` prop is set to the `id` property of each item, which ensures that each element in the list has a unique key.

It's important to note that the key should be unique within the list, not globally unique. If your data source has unique ids, it's recommended to use them as keys. If not, you can use the index of the element in the array as the key. However, it's not recommended to use the index as a key if the list can change dynamically, because it can lead to unexpected behavior.

React also provides a built-in hook `useState` that can be used to manage lists in functional components. This hook returns an array containing the current state and the function to update the state, the state updates can be used to update the list and also the key should be unique within the list.

In summary, Lists and Keys in React are used to render multiple elements of the same type and React requires that each element in the list has a unique key, this allows React to efficiently update the list and minimize the number of DOM operations, which improves performance.

What will happen if we use index as key in a list

Using the index of an element in an array as a key when rendering lists in React can lead to unexpected behavior and performance issues.

When React renders a list, it uses the keys to identify which elements have been added, removed, or changed. If you use the index as a key, React will assume that the element at a specific index is always the same element and will not re-render it when it changes. This can lead to stale data being displayed in the list, and can also cause performance issues if the list changes frequently.

Another problem with using the index as a key is that it can cause elements to move around in the list unexpectedly. This can happen when elements are added, removed, or reordered in the array. When this happens, React will re-render the entire list, causing all elements to lose their current state and re-mount, this can lead to unexpected behavior such as the component losing its focus or scroll position.

In summary, using the index as a key in a list can lead to unexpected behavior and performance issues. It's recommended to use unique keys for each element in a list, if the data source has unique ids, it's recommended to use them as keys. If not, you can use some unique properties of the object, such as an id or name, as the key.
Here's an example of how using the index as a key can lead to unexpected behavior:

```
function TodoList({ items }) {
  const [todos, setTodos] = useState(items);

  function handleDelete(index) {
    setTodos(todos.filter((_, i) => i !== index));
```

```
  }

  return (
    <ul>
      {todos.map((item, index) => (
        <li key={index}>
          {item}
          <button onClick={() => handleDelete(index)}>Delete</button>
        </li>
      ))}
    </ul>
  );
}
```

In this example, the TodoList component renders a list of todos and displays a delete button next to each item. When the delete button is clicked, the handleDelete function is called and it removes the item from the todos array by using Array.filter() method.

The problem with this code is that, it's using the index of the item as the key, since the index of an item in the array changes after an item is deleted, this will cause React to re-render the entire list, even though only one item has been removed.

As a result, when the delete button is clicked, all the list items will be re-rendered, causing them to lose their current state, such as focus or scroll position.

Here's an example of how to use unique keys to avoid this problem:

```
function TodoList({ items }) {
  const [todos, setTodos] = useState(items);

  function handleDelete(id) {
    setTodos(todos.filter(todo => todo.id !== id));
  }

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          {todo.text}
          <button onClick={() => handleDelete(todo.id)}>Delete</button>
        </li>
      ))}
    </ul>
  );
```

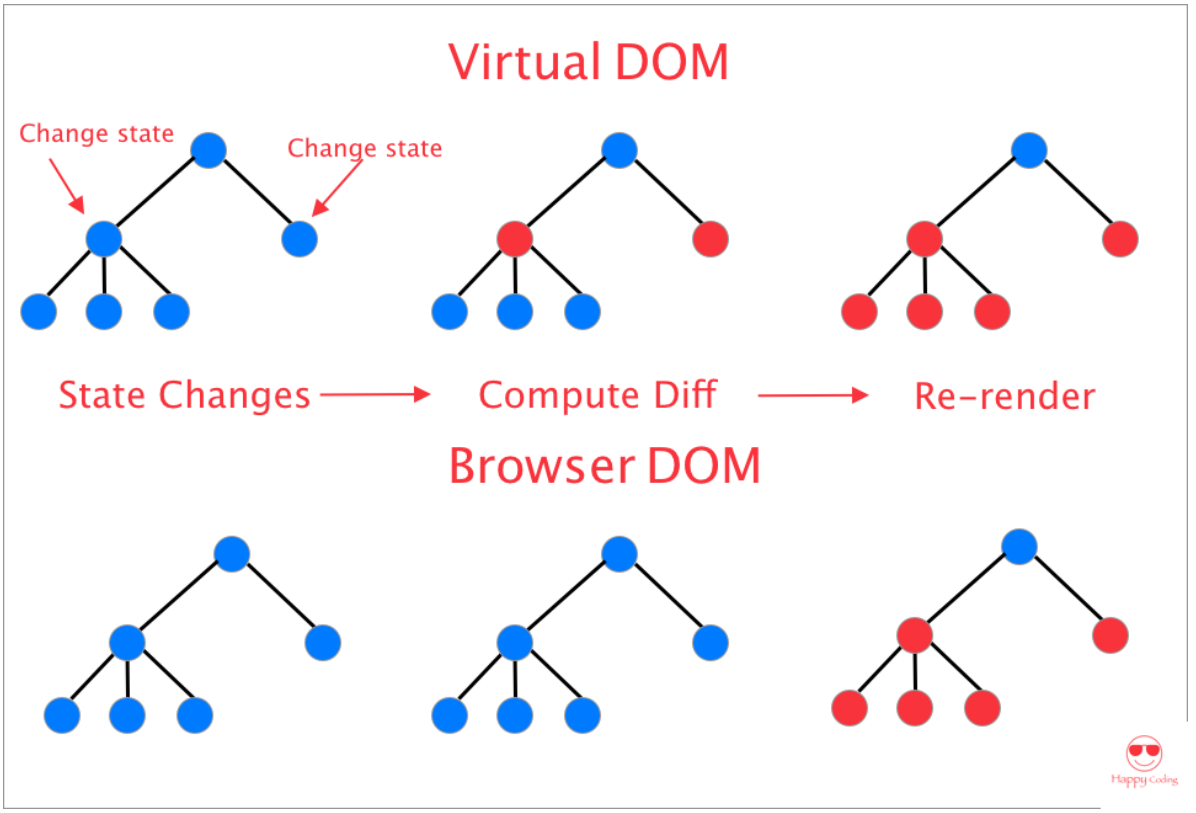
```
}
```

In this example, the `TodoList` component is using the id of the todo as the key, this ensures that each item in the list has a unique key, even after an item is deleted, React will be able to identify which item has been removed and update the list efficiently.

It's important to use a unique key for each item in a list, this ensures that React can efficiently update the list and minimize the number of DOM operations, which improves performance. And this will also avoid unexpected behavior such as the component losing its focus or scroll position.

Virtual DOM

The Virtual DOM (VDOM) is a mechanism used by React to improve the performance of web applications. It's a lightweight representation of the actual DOM (Document Object Model) that React uses to update the user interface efficiently.



When the state of a React application changes, React compares the current VDOM with the new VDOM, it then calculates the minimal set of changes required

to update the actual DOM to match the new VDOM. These changes are then made to the actual DOM, which updates the user interface.

This process of comparing the current VDOM with the new VDOM and then updating the actual DOM is called "reconciliation". Reconciliation is a process that happens behind the scenes and is handled by React automatically.

One of the main advantages of using the VDOM is that it allows React to update the user interface efficiently. By only updating the parts of the user interface that have changed, React reduces the number of DOM operations and improves the performance of the application. This is particularly beneficial for large and complex applications with many components.

Another advantage of using the VDOM is that it allows React to handle updates asynchronously. This means that React can batch multiple updates together and apply them all at once, which further improves performance.

In summary, Virtual DOM is a mechanism used by React to improve the performance of web applications, it's a lightweight representation of the actual DOM that React uses to update the user interface efficiently, React compares the current VDOM with the new VDOM, it then calculates the minimal set of changes required to update the actual DOM to match the new VDOM and apply those changes to the actual DOM, which updates the user interface. This process is called "reconciliation" and React handles it automatically.

Working with Forms

Working with forms in React can be a bit tricky, but with the right techniques it can be done easily.

There are two main ways to work with forms in React:

1. Controlled Components: In this method, React components handle the form state and update the form fields with the current state. This means that React is in control of the form data and the form fields are updated automatically when the state changes.

2. Uncontrolled Components: In this method, React components are not aware of the form state and the form fields are updated directly by the DOM. This means that the form data is not controlled by React and you need to manually handle form events and update the state.

Controlled Components:

A controlled component is a component that has its own state and updates the form fields with the current state. This means that React is in control of the form data and the form fields are updated automatically when the state changes.

Here's an example of a controlled component:

```
class Form extends React.Component {
  state = { name: "", email: "" };

  handleChange = event => {
    const { name, value } = event.target;
    this.setState({ [name]: value });
  };

  handleSubmit = event => {
    event.preventDefault();
    console.log(this.state);
  };

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input
            type="text"
            name="name"
            value={this.state.name}
            onChange={this.handleChange}
          />
        </label>
        <br />
        <label>
          Email:
          <input
            type="email"
            name="email"
            value={this.state.email}
            onChange={this.handleChange}
          />
        </label>
```

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

In this example, the `Form` component has its own state that contains the form data (`name` and `email`). The `handleChange` function updates the state when the form fields are changed and the `handleSubmit` function logs the form data to the console when the form is submitted.

Uncontrolled Components:

An uncontrolled component is a component that is not aware of the form state and the form fields are updated directly by the DOM. This means that the form data is not controlled by React and you need to manually handle form events and update the state.

Here's an example of an uncontrolled component:

```
class Form extends React.Component {
  handleSubmit = event => {
    event.preventDefault();
    const name = event.target.name.value;
    const email = event.target.email.value;
    console.log({ name, email });
  };

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" name="name" />
        </label>
        <br />
        <label>
          Email:
          <input type="email" name="email" />
        </label>
        <br />
        <button type="submit">Submit</button>
      </form>
```

```
    );
```

Formik

Formik is a popular library for handling forms in React. It provides a powerful way to handle forms with minimal code and helps to avoid some of the common pitfalls when working with forms in React.

Formik allows you to handle form state, validation, and submission in a centralized way. It also provides a way to handle form errors and provides utilities for working with form fields.

Here's an example of how to use Formik to create a simple form:

```
import { Formik, Form, Field } from "formik";

function MyForm() {
  return (
    <Formik
      initialValues={{ name: "", email: "" }}
      onSubmit={values => {
        console.log(values);
      }}
    >
      {({ errors, touched }) => (
        <Form>
          <Field name="name" placeholder="Name" />
          {errors.name && touched.name ? <div>{errors.name}</div> : null}
          <Field name="email" type="email" placeholder="Email" />
          {errors.email && touched.email ? <div>{errors.email}</div> : null}
          <button type="submit">Submit</button>
        </Form>
      )}
    </Formik>
  );
}
```

In this example, Formik is used to handle the form state, validation, and submission. The `initialValues` prop is used to set the initial form state and the `onSubmit` prop is used to handle the form submission.

The `<Field />` component is used to create form fields and the `name` prop is used to identify the field and link it to the form state.

Additionally, Formik provides an easy way to handle errors and validation, you can use the `errors` and `touched` props to check for errors and show them next to the form fields.

Formik provides many other features that can help in handling forms in React. Here are some of the additional features that Formik provides:

- Validation: Formik has built-in support for validation using Yup, a popular validation library. You can define a validation schema using Yup and Formik will automatically validate the form fields and provide the errors.
- Error handling: Formik makes it easy to handle form errors, you can use the errors and touched props to check for errors and show them next to the form fields. Formik also provides an ErrorMessage component that can be used to show the error message for a specific field.
- Field-level validation: Formik allows you to define custom validation logic for specific form fields. You can use the validate prop to define a validation function that will be called whenever the form field value changes.
- Custom form fields: Formik allows you to create custom form fields, you can use the Field component to create a custom form field that can be used throughout your application.
- Handling complex forms: Formik makes it easy to handle complex forms with multiple fields and nested data structures. You can use the values prop to access the form data and the setFieldValue function to update specific form fields.
- Controlling form state: Formik provides several props and methods that can be used to control the form state. For example, you can use the isSubmitting prop to disable the form submit button while the form is being submitted.

Overall, Formik provides a powerful and flexible way to handle forms in React. It makes it easy to handle form state, validation, and submission, and helps to avoid some of the common pitfalls when working with forms in React.

Session 3

Hooks

Hooks are a new feature in React that allow you to use state and other React features in functional components. Prior to hooks, state and other features such

as lifecycle methods were only available in class components.

Hooks are functions that allow you to "hook" into React's state and lifecycle features from within functional components. They make it possible to add state and other features to functional components without having to convert them to class components.

There are two main types of hooks:

1. State Hooks: useState allows you to add state to a functional component. It returns an array with two elements, the current state and a function to update the state.

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

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

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

2. Effect Hooks: useEffect allows you to run side effects, such as fetching data or setting up subscriptions, in functional components. It is similar to lifecycle methods in class components, such as componentDidMount and componentDidUpdate.

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

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

  useEffect(() => {
    fetch(`https://my-api.com/data/${id}`)
      .then(res => res.json())
      .then(setData);
  }, [id]);

  return (
    <>
      {data ? (
```

```
        <p>Data: {data.name}</p>
      ) : (
        <p>Loading...</p>
      )}
    </>
  );
}
```

Hooks are a powerful feature that allows you to write more readable, maintainable, and testable code in React. They also make it easy to reuse logic between different components.

Here are a few additional points about hooks in React:

- Hooks are only available in functional components: You cannot use hooks in class components or in regular JavaScript functions.
- Hooks are called at the top level of a component: Hooks should only be called at the top level of a component, and not inside loops or conditions.
- Hooks should only be called from React functions: Hooks should only be called from React functional components or custom hooks.
- Hooks let you reuse logic: Hooks allow you to extract logic from components and reuse it across different components. This makes your code more reusable and easier to test.
- Hooks only work inside React: Hooks only work inside React functions, they don't work in regular JavaScript functions or in other libraries or frameworks.

Overall, Hooks are a great addition to React that allows you to write more readable and maintainable code. They also make it easy to reuse logic between different components and to write tests for your components.

Lifecycle Methods

Lifecycle methods are methods that are called at specific points in the lifecycle of a React component. These methods are only available in class components and are automatically called by React at certain points in the component's lifecycle.

Here are some of the most common lifecycle methods in React:

- constructor(props): This method is called before the component is rendered. It is used to set up the initial state of the component and to bind any event handlers. The props passed to the constructor are the same props that are passed to the component when it is rendered.

- componentDidMount(): This method is called after the component is rendered to the DOM. It is commonly used to fetch data or set up subscriptions. This method is a good place to do any setup that requires the DOM.
- componentDidUpdate(prevProps, prevState): This method is called after the component updates. It is commonly used to handle side effects, such as updating the DOM or sending analytics data. The prevProps and prevState parameters contain the props and state of the component before the update.
- componentWillUnmount(): This method is called before the component is removed from the DOM. It is commonly used to clean up any subscriptions or other resources that were created in componentDidMount().
- shouldComponentUpdate(nextProps, nextState): This method is called before the component updates. It is commonly used to optimize performance by preventing unnecessary re-renders. It returns a boolean value indicating whether the component should update or not.
- getSnapshotBeforeUpdate(prevProps, prevState): This method is called before the component updates, but after the component's props or state are updated. It is commonly used to capture a snapshot of the component's state for later use.
- render(): This method is called to render the component. It returns the JSX that will be rendered to the DOM. It's the most important lifecycle method and should be as simple as possible.

It's important to note that not all lifecycle methods are used in every component. In fact, many components only use one or two lifecycle methods. It's important to understand the lifecycle methods available and when to use them, to write optimal and maintainable code.

Also, with the introduction of hooks, it's not necessary to use class components and lifecycle methods, but it's still important to know about these lifecycle methods to understand how a class component works and to be able to update or maintain older codebase.

Difference between Hooks and Lifecycle Method

Hooks and lifecycle methods are both used to control the behavior of React components, but they are used in different ways.

1. Hooks are used in functional components: Hooks are a way to add state and other features to functional components, whereas lifecycle methods are only available in class components. Hooks make it possible to use state and other features in functional components without having to convert them to class components.
2. Hooks are called at the top level of a component: Hooks should only be called at the top level of a component, and not inside loops or conditions. On the other hand, lifecycle methods are defined as a part of class component definition, and are called automatically by React during the component's lifecycle.
3. Hooks let you reuse logic: Hooks allow you to extract logic from components and reuse it across different components. This makes your code more reusable and easier to test. On the other hand, lifecycle methods are specific to a component and cannot be reused in other components.
4. Hooks are simpler to understand: Hooks are simpler to understand than lifecycle methods because they are just JavaScript functions and don't require the use of this keyword. They also make it easy to separate the logic of a component into smaller, more manageable parts.
5. Hooks provide a better way to handle state: Hooks provide a more intuitive way to handle state compared to lifecycle methods. The useState hook allows you to add state to a component in a way that is similar to how you would use a variable in JavaScript.
6. Hooks allow for better testing: Hooks allow for easier testing of components by allowing you to test logic as individual functions and not having to test and render a complete component.
7. Hooks provide better performance: Hooks allow for better performance by allowing you to control when and how often the component should update, instead of relying on React to decide for you.

In summary, Hooks and lifecycle methods are both important tools for managing the behavior of React components, but they are used in different ways. Hooks are used in functional components to add state and other features, whereas lifecycle methods are used in class components to control the behavior of a component during its lifecycle. Hooks provide a more modern and simpler way of handling component's state and lifecycle, however, lifecycle methods are still important to understand in order to maintain and update older codebase.

Different hooks in React

React provides a set of built-in hooks that you can use to add functionality to functional components. Here are some of the most commonly used hooks with code examples:

1. useState(initialValue): This hook allows you to add state to a functional component. It takes an initial value as an argument and returns an array with two elements: the current state and a function to update the state.

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

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

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

2. useEffect(callback, dependencies): This hook allows you to run side effects in a functional component, such as fetching data or updating the DOM. It takes a callback function and an array of dependencies as arguments. The callback function will run after the component renders. The dependencies are used to determine when the effect should run. If the dependencies haven't changed since the last render, the effect won't run.

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

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

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

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

```
    </>
  );
}
```

3. useContext(context): This hook allows you to access context in a functional component. It takes a context object as an argument and returns the current context value for that context.

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

const MyContext = React.createContext();

function Example() {
  const value = useContext(MyContext);

  return <p>Value: {value}</p>;
}
```

4. useReducer: The `useReducer` hook takes a reducer function and an initial state as arguments and returns the current state and a dispatch function to update the state. The reducer function takes the current state and an action as arguments, and returns the new state based on the action. The component can then use the state and dispatch function to handle state changes. In this example, the component has two buttons to increment and decrement the count, and the reducer function updates the state accordingly.

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

const initialState = { count: 0 };

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

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

```
    </>
  );
}
```

5. useCallback(callback, dependencies): This hook allows you to memoize a function so that it only re-renders if its dependencies have changed. It takes a callback function and an array of dependencies as arguments and returns the memoized function.

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

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

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

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

6. useMemo(callback, dependencies): This hook allows you to memoize a value so that it only re-calculates if its dependencies have changed. It takes a callback function that returns the value and an array of dependencies as arguments and returns the memoized value.

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

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

  const double = useMemo(() => count * 2, [count]);

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

7. useRef(initialValue): This hook allows you to create a reference to a value that persists across renders. It takes an initial value as an argument and returns an object with a current property that holds the current value of the reference.

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

function Example() {
  const inputRef = useRef(null);

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

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>Focus input</button>
    </>
  );
}
```

Each hook serves a specific purpose and can be used in different situations, understanding when to use which hook is an important part of mastering React development.

Different Lifecycle Methods

In React class-based components, there are several lifecycle methods that allow you to control when a component is created, updated, or destroyed. Here are some of the most commonly used lifecycle methods with code examples:

1. constructor(props): This method is called before the component is mounted. You can use the constructor to set the initial state of the component and bind event handlers.

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

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

```
      </>
    );
  }

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

2. componentDidMount(): This method is called after the component is mounted. You can use it to fetch data, create a timer, or add event listeners.

```
class Example extends React.Component {
  state = { count: 0 };

  componentDidMount() {
    document.title = `Count: ${this.state.count}`;
  }

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

3. shouldComponentUpdate(nextProps, nextState): This method is called before the component is updated. You can use it to decide whether or not to re-render the component. It takes the next props and next state as arguments and returns a boolean.

```
class Example extends React.Component {
  state = { count: 0 };

  shouldComponentUpdate(nextProps, nextState) {
    return nextState.count !== this.state.count;
  }

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

```
  }
}
```

4. componentDidUpdate(prevProps, prevState): This method is called after the component is updated. You can use it to handle side effects, update the DOM, or send a network request.

```
class Example extends React.Component {
  state = { count: 0 };

  componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
      console.log(`Count: ${this.state.count}`);
    }
  }

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

5. componentWillUnmount(): This method is called before the component is unmounted, which means that it is being removed from the DOM. You can use this method to cleanup any timers, event listeners, or network requests that you have set up in the componentDidMount() method.

```
class Example extends React.Component {
  state = { intervalId: null };

  componentDidMount() {
    const intervalId = setInterval(() => {
      console.log('tick');
    }, 1000);
    this.setState({ intervalId });
  }

  componentWillUnmount() {
    clearInterval(this.state.intervalId);
  }

  render() {
    return <p>Hello World</p>;
  }
}
```

6. componentDidCatch(error, info): This method is called if any of the component's children throws an error. You can use this method to handle errors, log them, or display an error message. For example, you could use this method to display a "Something went wrong" message to the user instead of crashing the whole application.

```
class Example extends React.Component {
  state = { hasError: false, error: null, info: null };

  componentDidCatch(error, info) {
    this.setState({ hasError: true, error, info });
  }

  render() {
    if (this.state.hasError) {
      return (
        <>
          <p>An error occurred: {this.state.error.toString()}</p>
          <pre>{this.state.info.componentStack}</pre>
        </>
      );
    }

    return this.props.children;
  }
}
```

It's important to note that the use of lifecycle methods is discouraged in recent versions of React. Instead hooks are encouraged to use.

Why are functional components preferred over class components ?

Functional components are preferred over class components for a few reasons:

1. Simplicity: Functional components are simpler to write and understand than class components. They are just JavaScript functions that take props as input and return JSX as output. This makes it easier to reason about what a component does and how it works.
2. Performance: Functional components are more performant than class components because they don't have the overhead of creating an instance and binding methods.

3. Reusability: Functional components are more reusable than class components. They don't have state or lifecycle methods, which means they don't have any hidden dependencies or side effects. This makes it easier to share, test, and reason about the logic.
4. Easy to understand: Functional components are easy to understand and reason about. Since they are just JavaScript functions, it is easy to understand what they do and how they work.
5. Scalability: Functional components are more scalable than class components. With hooks, you can extract stateful logic from a component, and keep it separate from the component itself. This can make it easier to share, test, and reason about the logic.
6. Hooks: Hooks are a new addition in React, they are only available in functional components, they allow stateful logic in functional components, making functional components more powerful, and allowing them to replace class components in most cases.

In summary, functional components are preferred over class components because they are simpler to write, understand, and reason about, more performant, reusable, easy to understand and scalable. They also allow the use of hooks.

Why are hooks preferred over lifecycle methods ?

Hooks are preferred over lifecycle methods for a few reasons:

1. Simplicity: Hooks allow you to use state and other React features in functional components, which makes it easier to write and understand the code. It eliminates the need to understand and remember the lifecycle methods and their order, especially when working with complex components.
2. Reusability: Hooks are reusable across different components, which allows for more code sharing and less duplication. This can make it easier to maintain and test the code.
3. Performance: With Hooks, you have more control over when a component re-renders, which can lead to better performance. This is because functional components with hooks will only re-render if the state or props passed to the Hook have changed.
4. Easy to understand: Hooks are easy to understand and reason about. Since they are just JavaScript functions, it is easy to understand what they do and how they work.

5. Scalability: With Hooks, you can extract stateful logic from a component, and keep it separate from the component itself. This can make it easier to share, test, and reason about the logic.

In summary, Hooks provide a more direct and simple approach to using state and other React features in functional components, making the code more readable, scalable, and maintainable.

Session 4

What is an API ?

An API, or Application Programming Interface, is a set of rules and protocols that allows different software systems to communicate with each other. It acts as an intermediary that allows two or more systems to talk to each other.

For example, imagine you're using a weather app on your phone. When you open the app, it needs to get the current weather information from somewhere. The weather app uses an API to communicate with a weather service, which provides it with the current temperature, humidity, and forecast. The weather app can then use this information to display the current weather on your screen.

In this example, the weather app acts as a client, which sends a request to the weather service, which acts as a server. The client sends a request through the API to the server, asking for the current weather information. The server receives the request, processes it and sends back a response with the requested information. The client then uses this information to display the current weather.

APIs are used to allow different systems to share data and functionality. This can be especially useful when different systems are created by different companies or organizations, or when different systems use different technologies.

APIs are widely used in the development of web and mobile applications, and in the integration of different systems. The most common types of APIs are REST and SOAP.

Working with API's

Working with APIs in ReactJS involves making HTTP requests to retrieve or send data to a server. There are several libraries that can be used for making HTTP requests in ReactJS such as axios, fetch, and superagent.

Fetch

Here's an example of how to use the fetch method to make a GET request to an API in a ReactJS functional component:

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

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

  return (
    <div>
      {data ? <p>Data: {JSON.stringify(data)}</p> : <p>Loading...</p>}
    </div>
  );
}
```

In this example, we use the `useState` hook to initialize a state variable `data` with `null` value. We use the `useEffect` hook to make the API call when the component is rendered. In the `useEffect` block, we define an async function `fetchData` that makes a GET request to the API using the `fetch` method.

The `fetch` method returns a promise that resolves to a `Response` object. We then call the `json()` method on the `Response` object to parse the response as a JSON object. We then set the data to the state variable `data` using the `setData` function. If there is an error, we catch it and log it to the console.

In the component's render method, we check if the data is available and render it or show a loading message.

It is important to note that, for security reasons, it is not recommended to use the `fetch` method directly in the browser as it may expose the API key, it is recommended to use a library like axios, superagent or any other library that allows you to add authentication headers.

Axios

Here's an example of how to use axios to make a GET request to an API in a ReactJS functional component:

```
import axios from 'axios';

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

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

  return (
    <div>
      {data ? <p>Data: {JSON.stringify(data)}</p> : <p>Loading...</p>}
    </div>
  );
}
```

In this example, we first import axios library. We then use the `useState` hook to initialize a state variable `data` with `null` value. We use the `useEffect` hook to make the API call when the component is rendered. In the `useEffect` block, we define an async function `fetchData` that makes a GET request to the API using the `axios.get` method.

The `await` keyword is used to wait for the response from the API. We then set the data to the state variable `data` using the `setData` function. The

response.data contains the data returned from the API. If there is an error, we catch it and log it to the console.

In the component's render method, we check if the data is available and render it or show a loading message.

It is important to note that, for security reasons, it is not recommended to use the `fetch` method directly in the browser as it may expose the API key, it is recommended to use a library like axios, superagent or any other library that allows you to add authentication headers.

With axios you can also add headers, authentication, and custom parameters for the request, also it has built-in error handling mechanism, you can also cancel request and intercept request and responses.

Superagent

Here's an example of how to use superagent to make a GET request to an API in a ReactJS functional component:

```
import request from 'superagent';

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

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

  return (
    <div>
      {data ? <p>Data: {JSON.stringify(data)}</p> : <p>Loading...</p>}
    </div>
  );
}
```

In this example, we first import the superagent library. We then use the `useState` hook to initialize a state variable `data` with `null` value. We use the `useEffect` hook to make the API call when the component is rendered. In the `useEffect` block, we define an async function `fetchData` that makes a GET request to the API using the `request.get` method.

The `await` keyword is used to wait for the response from the API. We then set the data to the state variable `data` using the `setData` function. The `response.body` contains the data returned from the API. If there is an error, we catch it and log it to the console.

In the component's render method, we check if the data is available and render it or show a loading message.

Superagent is also similar to axios and fetch, you can use it to make HTTP requests, it also allows to add headers, authentication and custom parameters for the request, it also has built-in error handling and you can also cancel request.

It is important to note that, for security reasons, it is not recommended to use the `fetch` method directly in the browser as it may expose the API key, it is recommended to use a library like axios, superagent or any other library that allows you to add authentication headers.

Best among fetch, axios and superagent and why ?

All three libraries (fetch, axios, and superagent) are popular choices for making HTTP requests in ReactJS and can be used effectively to work with APIs.

Fetch is a modern browser API that allows you to make HTTP requests. It is built-in to most modern browsers, so you don't need to install any additional libraries. However, it doesn't have a built-in mechanism for handling errors and it doesn't support older browsers. You also need to handle the promises to get the data.

Axios is a popular library for making HTTP requests in JavaScript. It has a similar API to fetch, but it also has some additional features such as built-in error handling, support for older browsers, and the ability to cancel requests. It also allows you to add headers, authentication, and custom parameters for the request.

Superagent is also similar to axios and fetch, it is a lightweight library for making HTTP requests in JavaScript, it also allows to add headers, authentication, and custom parameters for the request, also it has built-in error handling and you can also cancel request.

All three libraries have their own advantages and disadvantages. Fetch is built-in to most modern browsers and you don't need to install any additional libraries. But it doesn't have a built-in mechanism for handling errors and it doesn't support older browsers. Axios is a popular library that has additional features like built-in error handling, support for older browsers, and the ability to cancel requests. Superagent is also similar to axios and fetch with similar functionalities.

The best one depends on your use case, if you want a simple and lightweight solution, you can use fetch, if you want a more feature-rich library with built-in error handling and support for older browsers you can use axios or superagent.

Canceling request in axios

Here's an example of how to cancel a request in axios using the `CancelToken` feature:

```
import axios from 'axios';

let cancel;

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

  useEffect(() => {
    const fetchData = async () => {
      try {
        if (cancel) {
          cancel();
        }
        const CancelToken = axios.CancelToken;
        cancel = CancelToken.source();
        const response = await axios.get('https://api.example.com/data', {
          cancelToken: cancel.token,
        });
        setData(response.data);
      } catch (error) {
        if (axios.isCancel(error)) {
          console.log('Request canceled', error.message);
        } else {
          console.error(error);
        }
```

```
      }
    };
    fetchData();
    return () => {
      cancel();
    };
  }, []);

  return (
    <div>
      {data ? <p>Data: {JSON.stringify(data)}</p> : <p>Loading...</p>}
    </div>
  );
}
```

In this example, we first import axios library. We then use the `useState` hook to initialize a state variable `data` with `null` value. We use the `useEffect` hook to make the API call when the component is rendered. In the `useEffect` block, we define an async function `fetchData` that makes a GET request to the API using the `axios.get` method. Before making the request, we check if a request is already ongoing, if so, we cancel it by calling the `cancel` function.

We create a new `CancelToken` by calling `axios.CancelToken.source()` and assign the returned object to the `cancel` variable. We then pass the `cancel.token` as a config option to the axios.get method.

In the try-catch block, we check if the error is a cancel error by calling `axios.isCancel(error)` , if it is we log that the request has been canceled, if not we log the error.

We also make sure to return a cleanup function that calls the cancel function so that we can cancel the request when the component unmounts.

It's important to note that, the request will only be cancelled if the request has not yet been sent to the server, if the request has been sent but the response hasn't been received yet, it is not guaranteed that the request will be cancelled.

What is react-query ?

React-query is a library that makes it easier to work with data in a React application. It allows you to fetch, cache and update data from APIs in a simple and efficient way.

Imagine you have a to-do list application, you want to fetch the to-do list items from an API, once the data is fetched, you want to store it in a cache so that you don't have to fetch the same data again and again, and also you want to update the to-do list items when the user makes any changes. React-query makes it easy to handle all these tasks by providing a set of hooks that you can use in your React components.

With React-query, you can use a hook called `useQuery` to fetch data from an API, this hook automatically handles caching, and refreshing the data, you can also pass a query key that is unique to the request and the query will only re-run when the key changes.

Another hook called `useMutation` allows you to update the data on the server.

In summary, React-query is a library that makes it easier to work with data in a React application, it allows you to fetch, cache and update data from APIs in a simple and efficient way.
Here's an example of how to use `useQuery` and `useMutation` hooks from the React-query library to fetch and update data from an API:

```
import { useQuery, useMutation } from 'react-query';

function TodoList() {
  const { data, status } = useQuery('todoList', () =>
    fetch('https://my-api.com/todo')
      .then(res => res.json())
  );

  const [updateTodo, { status: updateStatus }] = useMutation(todo =>
    fetch('https://my-api.com/todo', {
      method: 'PUT',
      body: JSON.stringify(todo),
      headers: { 'Content-Type': 'application/json' },
    }).then(res => res.json())
  );

  return (
    <div>
      {status === 'loading' && <p>Loading...</p>}
      {status === 'error' && <p>Error</p>}
      {status === 'success' && (
        <ul>
          {data.map(todo => (
            <li key={todo.id}>
```

```
              {todo.text}
              <button onClick={() => updateTodo(todo)}>Update</button>
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}
```

In this example, we first import the `useQuery` and `useMutation` hooks from the React-query library. In the `TodoList` component, we use the `useQuery` hook to fetch the data from the API. The first argument of the `useQuery` is a unique key for the query, and the second argument is a function that returns a promise that will resolve with the data.

The `useQuery` hook returns an object that contains the data, status, and other information about the query. We destructure the data and status properties and use them to conditionally render the component.

We also use the `useMutation` hook to update the data on the server, the first argument of the `useMutation` is a function that takes the data as an argument, this function should return a promise that will resolve when the mutation is completed.

In the render method, we display the `Loading...` message when the status is "loading", an "Error" message when the status is "error" , and a list of to-do items when the status is "success". We also add a button for each to-do item that, when clicked, will update the to-do item by calling the `updateTodo` function.

Session 5

What are Routes ?

In React, routes refer to the different URLs or paths that a user can visit within a web application. The concept of routing is used to navigate between different components of an application, depending on the current URL.

What is react-router-dom ?

React Router DOM is a library that provides a set of components for client-side routing in React applications. It is a part of the React Router library, and it is designed to work with React's virtual DOM. React Router DOM provides a set of components that can be used to declaratively map URLs to components and handle client-side navigation.

Some of the key components provided by React Router DOM include:

- BrowserRouter: This component is used to wrap the entire application and provides the routing context for the application.
- Link: This component is used to create links that correspond to different routes.
- Route: This component is used to define a route and the corresponding component that should be rendered for that route.
- Switch: This component is used to define a set of routes and render the first route that matches the current URL.

React Router DOM is widely used in React application for client-side routing, it allows developers to declaratively map URLs to components and handle client-side navigation.

react-router-dom v5

Basic Routing

In this example we have 3 “pages” handled by the router: a home page, an about page, and a users page. As you click around on the different `<Link>`s, the router renders the matching `<Route>`.
Note: Behind the scenes a `<Link>` renders an `<a>` with a real `href`, so people using the keyboard for navigation or screen readers will still be able to use this app.

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

export default function App() {
  return (
```

```
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/users">Users</Link>
            </li>
          </ul>
        </nav>

        {/* A <Switch> looks through its children <Route>s and
            renders the first one that matches the current URL. */}
        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/users">
            <Users />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}

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

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

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

Nested Routing

This example shows how nested routing works. The route `/topics` loads the `Topics` component, which renders any further `<Route>` 's conditionally on the paths `:id` value.

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

export default function App() {
  return (
    <Router>
      <div>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/topics">Topics</Link>
          </li>
        </ul>

        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/topics">
            <Topics />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}

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

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

function Topics() {
  let match = useRouteMatch();

  return (
    <div>
      <h2>Topics</h2>

      <ul>
        <li>
          <Link to={`${match.url}/components`}>Components</Link>
        </li>
        <li>
          <Link to={`${match.url}/props-v-state`}>
            Props v. State
          </Link>
        </li>
      </ul>

      {/* The Topics page has its own <Switch> with more routes
          that build on the /topics URL path. You can think of the
          2nd <Route> here as an "index" page for all topics, or
          the page that is shown when no topic is selected */}
      <Switch>
        <Route path={`${match.path}/:topicId`}>
          <Topic />
        </Route>
        <Route path={match.path}>
          <h3>Please select a topic.</h3>
        </Route>
      </Switch>
    </div>
  );
}

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

Route Matchers

There are two route matching components: Switchand Route. When a <Switch>is rendered, it searches through its children<Route>elements to find one whose pathmatches the current URL. When it finds one, it renders that <Route>and

ignores all others. This means that you should put <Route>s with more specific (typically longer) paths **before**less-specific ones.

If no <Route>matches, the <Switch>renders nothing (null).

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

function App() {
  return (
    <div>
      <Switch>
        {/* If the current URL is /about, this route is rendered
            while the rest are ignored */}
        <Route path="/about">
          <About />
        </Route>

        {/* Note how these two routes are ordered. The more specific
            path="/contact/:id" comes before path="/contact" so that
            route will render when viewing an individual contact */}
        <Route path="/contact/:id">
          <Contact />
        </Route>
        <Route path="/contact">
          <AllContacts />
        </Route>

        {/* If none of the previous routes render anything,
            this route acts as a fallback.

            Important: A route with path="/" will *always* match
            the URL because all URLs begin with a /. So that's
            why we put this one last of all */}
        <Route path="/">
          <Home />
        </Route>
      </Switch>
    </div>
  );
}

ReactDOM.render(
```

```
  <Router>
    <App />
  </Router>,
  document.getElementById("root")
);
```

One important thing to note is that a `<Route path>` matches the **beginning** of the URL, not the whole thing. So a `<Route path="/">` will **always** match the URL. Because of this, we typically put this `<Route>` last in our `<Switch>`. Another possible solution is to use `<Route exact path="/">` which **does** match the entire URL.

useHistory()

The `useHistory` hook gives you access to the `history` instance that you may use to navigate.

```
import { useHistory } from "react-router-dom";

function HomeButton() {
  let history = useHistory();

  function handleClick() {
    history.push("/home");
  }

  return (
    <button type="button" onClick={handleClick}>
      Go home
    </button>
  );
}
```

useLocation()

The useLocationhook returns the locationobject that represents the current URL. You can think about it like a useStatethat returns a new locationwhenever the URL changes.

This could be really useful e.g. in a situation where you would like to trigger a new “page view” event using your web analytics tool whenever a new page loads, as in the following example:

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

function usePageViews() {
  let location = useLocation();
  React.useEffect(() => {
    ga.send(["pageview", location.pathname]);
  }, [location]);
}

function App() {
  usePageViews();
  return <Switch>...</Switch>;
}

ReactDOM.render(
  <Router>
    <App />
  </Router>
);
```

useParams()

`useParams` returns an object of key/value pairs of URL parameters. Use it to access `match.params` of the current `<Route>`.

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

function BlogPost() {
  let { slug } = useParams();
  return <div>Now showing post {slug}</div>;
}

ReactDOM.render(
  <Router>
    <Switch>
      <Route exact path="/">
```

```
        <HomePage />
      </Route>
      <Route path="/blog/:slug">
        <BlogPost />
      </Route>
    </Switch>
  </Router>
);
```

Redirect()

Rendering a `<Redirect>` will navigate to a new location. The new location will override the current location in the history stack, like server-side redirects (HTTP 3xx) do.

```
<Redirect to="/somewhere/else" />
```

react-router-dom v6

Routes

In version 6 of React Router DOM, the Routes component is used to define the different routes of the application. Here's an example of how you would use React Router DOM version 6 to set up routes in your application:

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

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="about" element={<About />} />
      </Routes>
    </Router>
  );
```

```
}

function Home() {
  return <h1>Home</h1>;
}

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

In this example, the `Link` component is used to create links that navigate to different parts of the application when clicked. The `Routes` component is used to define the different routes of the application. Inside `Routes` component, `Route` component is used to define the route and the `path` prop specifies the URL path that the route should match. The `element` prop is used to specify which component should be rendered when the route is active.

You can also use `children` prop instead of element to define a component that should be rendered when there are no matching routes.

React Router DOM version 6 also has features like support for dynamic routing with the use of `path` prop, `match` object passed to a component and programmatic navigation using the `navigate` function.

It's important to note that React Router DOM v6+ is more powerful, flexible and efficient than v5, it's recommended to use v6+ for any new projects.

Nested Route

In version 6 of React Router DOM, you can create nested routes by using the `Routes` component within another `Routes` component. Here's an example of how you would create nested routes in your application:

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

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link>
```

```
        <Link to="/about">About</Link>
        <Link to="/products">Products</Link>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="about" element={<About />} />
        <Route path="products" element={<ProductList />}>
            <Route path=":productId" element={<ProductDetails />} />
        </Route>
      </Routes>
    </Router>
  );
}

function Home() {
  return <h1>Home</h1>;
}

function About() {
  return <h1>About</h1>;
}

function ProductList() {
  return <h1>Product List</h1>;
}

function ProductDetails({ match }) {
  return <h1>Product ID: {match.params.productId}</h1>;
}
```

In this example, the `ProductList` component has nested route for `productId`, The `path` prop for each `Route` component is used to specify the URL path that the route should match. The `element` prop is used to specify which component should be rendered when the route is active.

The `match` object passed to the component will contain the current `path` and any `params` included in the path.

It's important to note that when creating nested routes, the order of the routes is important. The most specific routes should be declared first and more general routes should be declared later.

React Router DOM v6+ allows developers to create dynamic, nested and rich routing structures in a more declarative way.

useRoutes()

In React Router DOM version 6, `useRoutes` is a hook that allows you to programmatically render routes based on the current location of the application. It is useful for dynamic routing, where the routes are determined at runtime based on certain conditions. Here's an example of how you would use `useRoutes` in your application:

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

function App() {
  const routes = useRoutes([
    {
      path: "/",
      element: <Home />
    },
    {
      path: "about",
      element: <About />
    },
    {
      path: "products",
      element: <ProductList />,
      children: [
        {
          path: ":productId",
          element: <ProductDetails />
        }
      ]
    },
    {
      path: "*",
      element: <NotFound />
    }
  ]);

  return (
    <Router>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
        <Link to="/products">Products</Link>
      </nav>
      {routes}
    </Router>
```

```
  );
}

function Home() {
  return <h1>Home</h1>;
}

function About() {
  return <h1>About</h1>;
}

function ProductList() {
  return <h1>Product List</h1>;
}

function ProductDetails({ match }) {
  return <h1>Product ID: {match.params.productId}</h1>;
}

function NotFound() {
  return <h1>404 - Not Found</h1>;
}
```

In this example, `useRoutes` is passed an array of route objects containing the paths and elements that should be rendered for each route. Each route object can also contain a `children` property for nested routes.

The `useRoutes` hook returns the element that should be rendered for the current location, which is then rendered inside the `Router` component.

`useRoutes` is a more advanced feature of React Router DOM v6, and it gives you more control over how routes are rendered in your application.

useSearchParams()

In React Router DOM version 6, `useSearchParams` is a hook that allows you to access and manipulate the URL search parameters (also known as query parameters) in your application.

Here's an example of how you would use `useSearchParams` in your application:

```
import { useSearchParams } from "react-router-dom";
```

```
function SearchPage() {
  const searchParams = useSearchParams();

  const [searchTerm, setSearchTerm] = useState(searchParams.get("q") || "");
  const [resultsPerPage, setResultsPerPage] =
useState(searchParams.get("resultsPerPage") || 10);

  function handleSearchSubmit(e) {
    e.preventDefault();
    searchParams.set("q", searchTerm);
    searchParams.set("resultsPerPage", resultsPerPage);
  }

  return (
    <form onSubmit={handleSearchSubmit}>
      <label>
        Search Term:
        <input
          type="text"
          value={searchTerm}
          onChange={e => setSearchTerm(e.target.value)}
        />
      </label>
      <label>
        Results Per Page:
        <input
          type="number"
          value={resultsPerPage}
          onChange={e => setResultsPerPage(e.target.value)}
        />
      </label>
      <button type="submit">Search</button>
    </form>
  );
}
```

In this example, `useSearchParams` is used to get the current search parameters from the URL. It returns an object that has methods to get, set or delete the search parameters.

The values of the search parameters can be accessed by calling the `get` method on the returned object, passing the name of the parameter as an argument.

The values of the search parameters can be updated by calling the `set` method on the returned object, passing the name of the parameter and its new value as arguments.

The values of the search parameters can be deleted by calling the `delete` method on the returned object, passing the name of the parameter as an argument.

In the above example we are using the `useState` hook to manage the state of the form fields and we are updating the search parameters when the form is submitted.

It's important to note that the changes made to the search parameters using this hook will update the URL and it will change the state of the application.

useParams()

In React Router DOM version 6, `useParams` is a hook that allows you to access the dynamic parts of the current route's path in your application.

Here's an example of how you would use `useParams` in your application:

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

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

  return (
    <div>
      <h1>User Profile</h1>
      <p>User ID: {id}</p>
    </div>
  );
}
```

In this example, `useParams` is used to get the value of the `id` parameter from the current route's path. The returned object has properties that correspond to the dynamic parts of the path.

In the above example, `id` is the dynamic part of the path and we are destructuring the returned object to get the value of `id` and use it to display the user's profile.

You can also use `useParams` with multiple dynamic parts in the path, like this:

```
function ProductDetails() {
  const { productId, variantId } = useParams();

  return (
    <div>
      <h1>Product Details</h1>
      <p>Product ID: {productId}</p>
      <p>Variant ID: {variantId}</p>
    </div>
  );
}
```

In this example, the `productId` and `variantId` are dynamic parts of the path and we are destructuring the returned object to get the values of both and use them to display the product's details.

It's important to note that `useParams` only works inside the components that are rendered by a `Routes` component, and that the keys of the returned object correspond to the names of the dynamic parts of the path defined in the `path` prop of the `Route` component.

Outlet

CustomRoutes.js

```
import React from "react";

import { Routes, Route, Link, useRoutes } from "react-router-dom";
import Book from "../Book";
import BookList from "../BookList";
import BooksLayout from "../BooksLayout";
import Home from "../Home";
import NewBook from "../NewBook";
import NotFound from "../NotFound";
import ForgotPassword from "./ForgotPassword";
import Login from "./Login";
import Private from "./Private";
import Restricted from "./Restricted";

const bookList = [
  {
    id: 1,
    name: "Arms And The Man",
    author: "George Bernard Shaw",
  },
```

```
  {
    id: 2,
    name: "Merchant of Venice",
    author: "William Shakespeare",
  },
];

const CustomRoutes = () => {
  const isUserAuthenticated = false;
  return (
    <div>
      <Routes>
        <Route
          path="/"
          element={<Restricted isUserAuthenticated={isUserAuthenticated} />}
        >
          {" "}
          {/*Restricted Routes*/}
          <Route path="login" element={<Login />} />
        </Route>
        <Route
          path="/"
          element={<Private isUserAuthenticated={isUserAuthenticated} />}
        >
          {" "}
          {/*Private Routes*/}
          <Route index element={<Home />} />
          <Route path="/books" element={<BooksLayout bookList={bookList} />}>
            <Route index element={<BookList />} />
            <Route path=":id" element={<Book />} />
          </Route>
        </Route>

        <Route path="*" element={<NotFound />} />
      </Routes>
    </div>
  );
};

export default CustomRoutes;
```

Private.js

```
import React from "react";
import { Outlet, Navigate } from "react-router-dom";

const Private = (props) => {
  return (
    <div>
      {props.isUserAuthenticated ? <Outlet /> : <Navigate to="/login" />}
```

```
    </div>
  );
};

export default Private;
```

Restricted.js

```
import React from "react";

import { Outlet, Navigate } from "react-router-dom";

const Restricted = (props) => {
  return (
    <div>{props.isUserAuthenticated ? <Navigate to="/" /> : <Outlet />}</div>
  );
};

export default Restricted;
```

BooksLayout.js

```
import React from "react";
import { Outlet, Link } from "react-router-dom";

const obj = { hello: "worlds" };

const BooksLayout = (props) => {
  return (
    <div>
      <nav>
        <li>
          {" "}
          <Link to="/books/1">Book 1</Link>
        </li>
        <li>
          {" "}
          <Link to="/books/2">Book 2</Link>
        </li>
        {/* <li>
          {" "}
          <Link to="/books/new">New Book</Link>
        </li> */}
      </nav>
      <Outlet context={{ bookList: props.bookList }} />
    </div>
  );
};
```

```
export default BooksLayout;
```

BookList.js

```
import React from "react";

import { Link, useOutletContext } from "react-router-dom";

const BookList = () => {
  const context = useOutletContext();

  return (
    <div>
      {context.bookList.map((book) => {
        return (
          <div key={book.id}>
            <h2>NAME : {book.name}</h2>
            <h3>AUTHOR : {book.author}</h3>

            {/* <button>
              <Link to={`/books/${book.id}`}>Show</Link>
            </button> */}
            <hr />
          </div>
        );
      })}
    </div>
  );
};

export default BookList;
```

Book.js

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

import { useParams, useOutletContext } from "react-router-dom";

const Book = () => {
  const [selectedBook, setSelectedBook] = useState({});
  const params = useParams();
  const context = useOutletContext();

  useEffect(() => {
    // const fetchedbook = context.bookList
    //   .map((book) => {
    //     if (book.id === parseInt(params.id)) {
    //       return book;
    //     }
```

```
    //   })
    //   .filter(function (element) {
    //     return element !== undefined;
    //   });

    const fetchedbook = context.bookList.filter(
      (book) => book.id === parseInt(params.id)
    );

    setSelectedBook(fetchedbook[0]);
  }, []);

  return (
    <div>
      <h1>Book Detail</h1>
      <h2>NAME : {selectedBook?.name}</h2>
      <h3>AUTHOR: {selectedBook?.author}</h3>
    </div>
  );
};

export default Book;
```

Higher Order Component (HOC)

A Higher Order Component (HOC) is a component that takes another component as an input and returns a new component with additional functionality. It is a way to reuse component logic.

A common use case for HOCs is to add additional functionality such as authentication or authorization to a component.

Here's an example of a HOC that adds authentication functionality to a component:

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

const withAuth = (WrappedComponent) => {
  return class extends Component {
    componentDidMount() {
      if (!localStorage.getItem("isLoggedIn")) {
        this.props.history.push("/login");
      }
    }

    render() {
      return <WrappedComponent {...this.props} />;
```

```
    }
  };
};

export default withAuth;
```

In this example, the `withAuth` HOC takes a component as an input (`WrappedComponent`) and returns a new component that has the additional functionality of checking if the user is logged in. If the user is not logged in, the component will redirect the user to the login page.

To use the HOC, you can wrap your component with the HOC like this:

```
import withAuth from "./withAuth";

class ProfilePage extends React.Component {
  render() {
    return <div>Profile</div>;
  }
}

export default withAuth(ProfilePage);
```

In this example, the `ProfilePage` component is wrapped with the `withAuth` HOC, so it will now have the additional functionality of checking if the user is logged in before rendering the component.

You can also pass props to HOC and use it in Wrapped component

```
const withAuth = (WrappedComponent) => {
  return class extends Component {
    render() {
      return <WrappedComponent {...this.props} isAuth={true} />;
    }
  };
};
```

Here `isAuth` is a props that we are passing to `WrappedComponent`

HOCs are also useful for performance optimization, such as only re-rendering a component when certain props or state have changed.

It is important to note that HOCs are a way to reuse component logic and should not be used to change the visual output of a component. If you want to change the visual output of a component, you should use a technique called "render props" instead.

Session 6

Lifting State Up

"Lifting state up" is a technique used in React when components need to share state. It involves moving the shared state to a common ancestor component and passing the state and methods to update the state down to the child components as props.

For example, let's say you have two components, `ComponentA` and `ComponentB`, that need to share the same state of a counter. Here is the code for `ComponentA`:

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

const ComponentA = () => {
  const [counter, setCounter] = useState(0);

  const handleIncrement = () => {
    setCounter(counter + 1);
  }

  return (
    <>
      <button onClick={handleIncrement}>Increment</button>
      <p>Counter: {counter}</p>
    </>
  );
}

export default ComponentA;
```

`ComponentB` is similar but with some other functionality

To share the state of the counter between `ComponentA` and `ComponentB`, we can lift the state and the methods to update the state up to a common ancestor

component. In this case, let's say that the common ancestor component is called `Parent`.

```
import React, { useState } from 'react';
import ComponentA from './ComponentA';
import ComponentB from './ComponentB';

const Parent = () => {
  const [counter, setCounter] = useState(0);

  return (
    <>
      <ComponentA counter={counter} setCounter={setCounter} />
      <ComponentB counter={counter} setCounter={setCounter} />
    </>
  );
}

export default Parent;
```

In this example, `Parent` component is the common ancestor of both `ComponentA` and `ComponentB`, so it has the state of the counter and the methods to update the counter. The state and methods are passed down to `ComponentA` and `ComponentB` as props.

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

const ComponentA = ({counter, setCounter}) => {
    const handleIncrement = () => {
    setCounter(counter + 1);
  }

  return (
    <>
      <button onClick={handleIncrement}>Increment</button>
      <p>Counter: {counter}</p>
    </>
  );
}

export default ComponentA;
```

In this way, both `ComponentA` and `ComponentB` have access to the same state of the counter and can update the counter by calling the `setCounter` method

passed down as props.

It is important to note that Lifting state up is not always the best solution. If the components are not closely related, it may be better to use a state management library like Redux to handle the shared state.

State Management

State management in React refers to the process of storing and maintaining the state of a component or application, and passing that state to other components as needed. State can include any data or information that a component needs to function and render properly, such as user input, component or page state, or application data.

In React, state can be managed in a few different ways:

- Using the state property on class components or the useState hook on functional components. This allows for local state management within a single component, but can become complex and hard to maintain as the component or application grows.
- Using a centralized state management library such as Redux or MobX. These libraries provide a way to store and manage global state, and make it easy to update state and pass it down to other components through the use of actions, reducers and selectors.
- Using the Context API to share state between components that are not directly related in the component tree. This can be useful for sharing global state without the added complexity of a state management library.

State management is an important aspect of building React applications, as it allows for dynamic and interactive user interfaces, and can also be used to handle and manage application data, such as API calls and user authentication.

Zustand

Zustand is a lightweight state management library for React that allows you to store and manage the state of your application in a simple and efficient way. It is based on the popular state management library Redux, but is designed to be simpler and more intuitive to use.

Zustand allows you to create a global store of state, which can be accessed and updated by any component in your application. This store is created using the `create` method provided by the library, and is passed down to your components using the `useStore` hook.

Here is an example of a more complex use case of Zustand:

```
import { create } from 'zustand';

const [useStore] = create((set) => {
  const initialState = {
    user: null,
    isLoading: false,
    error: null,
  };

  return {
    ...initialState,
    setUser: (user) => set((state) => ({ ...state, user })),
    setLoading: (isLoading) => set((state) => ({ ...state, isLoading })),
    setError: (error) => set((state) => ({ ...state, error })),
    clearError: () => set((state) => ({ ...state, error: null })),
    login: async (email, password) => {
      try {
        setLoading(true);
        const response = await axios.post('/login', { email, password });
        setUser(response.data);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    },
    logout: () => {
      setUser(null);
    },
  };
});

function LoginPage() {
  const { user, isLoading, error, login, clearError } = useStore();

  return (
    <div>
      {user ? (
        <div>Welcome, {user.name}! <button onClick={logout}>Logout</button></div>
      ) : (
        <form onSubmit={(e) => {
```

```
            e.preventDefault();
            login(email,password);
          }>
          <input
            type="email"
            placeholder="Email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          />
          <input
            type="password"
            placeholder="Password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
          />
          <button type="submit" disabled={isLoading}>
            {isLoading ? 'Loading...' : 'Login'}
          </button>
          {error && <div onClick={clearError}>{error.message}</div>}
        </form>
      )}
    </div>
  );
}
```

In this example, we have a global store that holds the state of the login page. The store holds the user's data, a loading state, and an error state. We also have several actions in the store such as `login`, `logout`, `setUser`, `setLoading`, `setError` and `clearError`.

The `LoginPage` component uses the `useStore` hook to access the state and actions from the global store. When the form is submitted, it calls the `login` action, which makes an API call to the server to perform the login. If the login is successful, it updates the `user` state, otherwise it updates the `error` state.

The example also show how you can use `setLoading` and `setError` to handle loading and error state and how you can use `clearError` to clear the error state.

React Context API

The React Context API is a way to share state between components in a React application without having to pass props down through multiple levels of components. It allows you to create a "context" that can be consumed by any component in your application, regardless of its location in the component tree. Here's an example of how the React Context API can be used to manage the user authentication state in a more complex application:

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

// Create the context
const AuthContext = createContext();

// The provider component
function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  function login(user) {
    setUser(user);
  }

  function logout() {
    setUser(null);
  }

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

// A component that consumes the context
function Header() {
  const { user, logout } = useContext(AuthContext);
  return (
    <header>
      <nav>
        {user ? (
          <>
            <span>Hello, {user.name}</span>
            <button onClick={logout}>Logout</button>
          </>
        ) : (
          <Link to="/login">Login</Link>
        )}
      </nav>
    </header>
  );
}
```

```
function App() {
  return (
    <AuthProvider>
      <Header />
      <Switch>
        <Route path="/login" component={Login} />
        <Route path="/dashboard" component={Dashboard} />
      </Switch>
    </AuthProvider>
  );
}
```

In this example, the `AuthProvider` component wraps the entire application and holds the user authentication state. The `Header` component is able to consume the user state and the `login` and `logout` functions from the context, so it can display different content depending on whether the user is logged in or not. Any other components in the application that need to access the user state or the `login` and `logout` functions can also consume them using the `useContext` hook.

It's worth noting that context is a powerful tool, and it can be a good way to share state between components without having to pass props down through multiple levels of the component tree. However, it should be used with caution, as it can make your codebase harder to reason about if it's not used correctly.

React Context API vs Zustand

React Context API and Zustand are both libraries that can be used for state management in React, but they have some differences.

React Context API is a built-in feature of React that allows you to share state between components without having to pass props down through multiple levels of the component tree. It is created using the `createContext` function, and the state is managed using a `Provider` component that wraps the entire application and a `useContext` hook that can be used by any component to consume the state.

On the other hand, Zustand is a third-party library that provides a similar way to manage state in React, but it has some additional features that make it more powerful and flexible. For example, Zustand allows you to create multiple stores for different parts of your application, it has an `immer`-based API that makes it

easy to update the state, and it provides a way to handle async actions, like data fetching.

In terms of when to use each library, it depends on the requirements of your application. If you only need to share a small amount of state between a few components, and don't need any advanced features like async actions, then React Context API might be a good fit. On the other hand, if you need to manage more complex state, or you want to make it easy to test and debug your state, then Zustand might be a better choice.

Session 7

React Composition Model

The React Composition Model is a way of building reusable components in React by combining smaller, simpler components together. It's a design pattern that promotes building your application as a collection of small, focused components that can be composed together to create more complex components.

Here's an example of how you can use the Composition Model to build a simple "Card" component:

```
import React from 'react';

function Card({ title, children }) {
  return (
    <div>
      <h2>{title}</h2>
      <div>{children}</div>
    </div>
  );
}

function App() {
  return (
    <Card title="Hello World">
      <p>This is some content inside the card.</p>
    </Card>
  );
}
```

In this example, the `Card` component is a simple function that takes in a `title` prop and some `children` and renders a div with a title and the children. The `App` component is then using the `Card` component and passing a title and some content as children.

You can also see the composition model in action by creating small focused components and using them in a more complex component:

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

function CardBody({ children }) {
  return <div>{children}</div>;
}

function Card({ title, children }) {
  return (
    <div>
      <CardHeader title={title} />
      <CardBody>{children}</CardBody>
    </div>
  );
}
```

This approach allows for a more modular and reusable codebase, by breaking down the application into smaller, focused and testable components, making it easy to reuse and reason about the code.
Sure, here's a more detailed and different example of the React Composition Model in action. Let's say we're building a social media application and we have a `Post` component that displays a user's post along with the user's name, profile picture, and the number of likes on the post.

We can start by breaking down the `Post` component into smaller, simpler components:

```
function Post({ post }) {
  return (
    <div>
      <UserInfo user={post.user} />
      <PostContent post={post} />
      <PostActions post={post} />
```

```
    </div>
  );
}

function UserInfo({ user }) {
  return (
    <div>
      <img src={user.profilePic} alt={user.name} />
      <h3>{user.name}</h3>
    </div>
  );
}

function PostContent({ post }) {
  return <p>{post.content}</p>;
}

function PostActions({ post }) {
  return <p>Likes: {post.likes}</p>;
}
```

Here, we have broken down the `Post` component into three smaller components: `UserInfo`, `PostContent`, and `PostActions`. Each of these components is focused on displaying a specific piece of information about the post and can be reused across the application.

By passing the `post` object as a prop to these smaller components, we are able to access the data we need to display the post in a modular and reusable way. Also, each component is simple and can be tested independently.

Also, this approach allows for flexibility and ease of maintenance, for example, if we want to add a new feature like the number of shares, we can create a new component `PostShares` and compose it into `Post` component.

You can also use hooks and functional components in above example as well.

Session 8

Redux

Redux is a JavaScript library for managing application state. It is often used in conjunction with React to manage the state of a React application, but it can be used with other JavaScript libraries or frameworks as well.

The core concept of Redux is that all application state is stored in a single, immutable "store." This store is essentially a plain JavaScript object that holds all the data for your application.

Changes to the store are made by dispatching "actions" to the store. An action is simply an object that describes the change you want to make. For example, an action might look like this:

```
{
  type: "ADD_TODO",
  payload: {
    text: "Write a Redux tutorial"
  }
}
```

When an action is dispatched, it is passed through a "reducer" function. A reducer is a pure function that takes the current state of the store and the action being dispatched, and returns the new state of the store. The reducer function is responsible for updating the store in response to the action.

Redux also allows the use of "middleware" which can be used for performing additional logic before the action reaches the reducer function. The middleware can be used for logging, error handling, and other tasks.

Overall, Redux provides a predictable and maintainable way of managing application state, by keeping the state in a centralized place, and allowing the use of actions and reducers to update the state in a controlled manner.

Why is Redux used ?

Redux is used to manage the state of a JavaScript application in a predictable and maintainable way.

There are several reasons why developers might choose to use Redux:

- Centralized state management: With Redux, all application state is stored in a single, immutable "store." This makes it easy to see and understand the entire state of an application at any given time, and it makes it easy to debug and troubleshoot issues.

- Predictable state updates: Redux uses a strict set of rules for how state can be updated. Actions are the only way to update the store, and actions must be handled by reducers, which are pure functions that take the current state and an action and return the new state. This makes it easy to understand what will happen when an action is dispatched and to debug any issues that arise.
- Easy to test: Because reducers are pure functions, they are very easy to test. You can write tests that pass in a specific state and action and check that the correct new state is returned.
- Middleware: With Redux, it's easy to add middleware in your application to perform additional logic before the action reaches the reducer function. This can be used for logging, error handling, and other tasks.
- Works well with React: Redux is often used in conjunction with React to manage the state of a React application. Redux's centralized state management and predictable state updates work well with React's declarative component model.

Overall, Redux provides a predictable and maintainable way of managing application state, by keeping the state in a centralized place, and allowing the use of actions and reducers to update the state in a controlled manner.

How does it solve prop-drilling ?

Prop-drilling refers to the process of passing props down through multiple levels of components in a React application in order to make them available to a child component that needs them. As the number of nested components increases, this can become unwieldy and difficult to manage.

Redux solves this problem by providing a centralized store for the application state. Instead of passing props down through multiple levels of components, components can simply connect to the store and access the state they need. This reduces the amount of props that need to be passed down, making the component tree more manageable.

To connect a component to the store, we use the connect function from the react-redux library. This function takes two arguments:

- a mapStateToProps function that maps the state in the store to the props the component needs

- a mapDispatchToProps function that maps the dispatch function to the props the component needs to update the state

This allows the component to access the state and dispatching actions directly from the store, without the need to pass down props through the component tree.

By using Redux, the component can directly access the state from the store, without the need of passing down the state through multiple levels of components, which is known as prop-drilling.

How does it go well with react ?

Redux and React are often used together because they complement each other well.

React is a library for building user interfaces using a component-based architecture. It allows developers to break down complex UI into small, reusable components that can be easily composed to create a larger UI. This makes it easy to reason about the component tree and manage the state of the application.

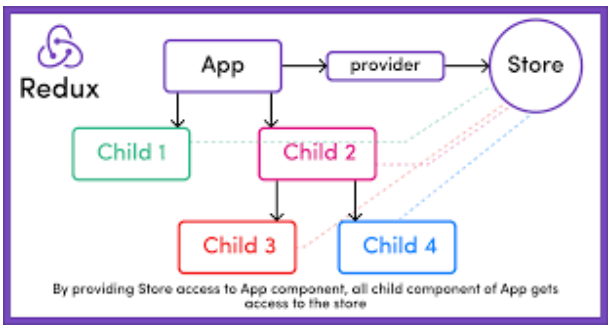
Redux, on the other hand, is a library for managing the state of the application in a centralized way. It provides a predictable way to handle updates to the state, allowing developers to easily track changes and debug the application.

When used together, React and Redux allow developers to build complex, large-scale applications in a manageable way. React provides the component-based architecture, while Redux provides the centralized state management.

Additionally, React-Redux library is provided which allows easy integration of redux with react, it provides a connect() function which can be used to connect the component to the store. This allows the component to access the state from the store and dispatch actions directly from the component.

Overall, React and Redux work well together by providing a powerful combination of component-based architecture and centralized state management. This allows developers to build complex, large-scale applications that are easy to reason about and maintain.

What is a Redux Store ?



In Redux, the store is the central location where the state of the application is stored. It holds the current state of the application and provides methods for updating the state and subscribing to state changes.

The store is created by passing the root reducer function and an initial state to the createStore() function provided by the redux library. The root reducer function takes the current state and an action as arguments and returns the next state.

The store has the following important methods:

- getState(): Returns the current state of the store.
- dispatch(action): Dispatches an action to the store. The action is passed to the root reducer function, which updates the state accordingly.
- subscribe(listener): Allows components to subscribe to state changes. The listener function is called every time the state is updated.

```
import { createStore } from 'redux';

// the root reducer function
const rootReducer = (state, action) => {
    switch(action.type) {
        case 'INCREMENT':
            return { ...state, count: state.count + 1 }
        case 'DECREMENT':
            return { ...state, count: state.count - 1 }
        default:
            return state;
    }
}

// the initial state
const initialState = { count: 0 }

// create the store
const store = createStore(rootReducer, initialState);
```

```
console.log(store.getState()) // { count: 0 }

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()) // { count: 1 }

store.dispatch({ type: 'DECREMENT' });
console.log(store.getState()) // { count: 0 }
```

In this example, the store is created using the createStore() function, and passed the rootReducer and the initial state, the store can then be used to dispatch actions and update the state.

1. Create a file named index.js and copy the provided code into it.
2. In the command line, run `npm install redux`. This will install the `redux` library, which is required to create the store.
3. Run `node index.js` in the command line. This will execute the code and show the initial state, then the state after each action.

What are Redux Actions ?

In Redux, actions are JavaScript objects that represent an event that has occurred in the application. They are the only way to trigger a change in the store, and they are the source of truth for what has happened in the application.

Actions are plain JavaScript objects that must have a `type` property, which describes the type of action that has occurred. They can also include additional data, called payload, that provides more information about the action.

Actions are created and dispatched by the application, and are then passed to the Redux store. The store will then use the action's `type` property to determine how to update the application's state.

Here is an example of a simple action in Redux:

```
const ADD_TODO = 'ADD_TODO';

const addTodo = (text) => ({
  type: ADD_TODO,
  payload: {
    text
  }
});
```

In the above example, `addTodo` is a function that creates an action object with a type of `ADD_TODO` and a payload of `text`. This action can then be dispatched to the store, which will use the `type` property to determine how to update the state of the application.

What are Redux Reducers ?

In Redux, reducers are functions that take in the current state of the application and an action, and then return a new state based on that action. They are used to update the application's state in response to actions that have been dispatched to the store.

A reducer function takes two arguments: the current state of the application, and an action object that was dispatched to the store. The function then examines the `type` property of the action object to determine what kind of action occurred, and updates the state accordingly.

Here's an example of a simple reducer function in Redux:

```
const initialState = {
  todos: []
}

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return {...state, todos: [...state.todos, action.payload]}
    case 'REMOVE_TODO':
      return {...state, todos: state.todos.filter(todo => todo.id !==
action.payload)}
    default:
      return state;
  }
};
```

In the above example, the `todoReducer` function takes in the current state of the application and an action object. It then uses a switch statement to examine the `type` property of the action object to determine what kind of action occurred. Based on the type of action, it will update the state by adding new todo or removing existing one.

It is important to note that the reducer functions should be pure functions, meaning that they should not have any side effects and should not mutate the state. Instead, they should create and return a new state object based on the current state and the action.

How to dispatch a Redux Action ?

To dispatch a Redux action, you can use the `dispatch()` function provided by the store. The `dispatch()` function takes a single argument, which is the action object that you want to dispatch.

Here's an example of how you can dispatch an action in a React component:

```
import { useDispatch } from 'react-redux';

function TodoForm({ addTodo }) {
  const dispatch = useDispatch();

  const handleSubmit = (event) => {
    event.preventDefault();
    const formData = new FormData(event.target);
    const todoText = formData.get('todoText');
    dispatch(addTodo(todoText));
  }

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

In the above example, the `TodoForm` component is using the `useDispatch` hook from the `react-redux` library to get access to the `dispatch` function from the store. Then, it's calling `dispatch(addTodo(todoText))` in the `handleSubmit` function to dispatch an action with the text of the todo.

It is important to note that the `addTodo` action creator should be imported and connected to the store before calling it in the component.

mapDispatchToProps()

`mapDispatchToProps` is a function that is used to map the `dispatch` function of the Redux store to the props of a component. This allows the component to dispatch actions and update the state of the store.

Here is an example of how `mapDispatchToProps` is used to map the `dispatch` function to the `onClick` prop of a button component:

```
import { connect } from 'react-redux';
import { incrementCounter } from './actions';

const mapDispatchToProps = (dispatch) => {
  return {
    onClick: () => dispatch(incrementCounter())
  }
}

const Button = ({ onClick }) => (
  <button onClick={onClick}>
    Increment
  </button>
);

export default connect(null, mapDispatchToProps)(Button);
```

In this example, the `mapDispatchToProps` function is used to map the `dispatch(incrementCounter())` function to the `onClick` prop of the `Button` component. When the button is clicked, the `incrementCounter` action is dispatched and the state is updated.

mapStateToProps()

`mapStateToProps` is a function that is used to map the state of the Redux store to the props of a component. This allows the component to access and display the state of the store.

Here is an example of how `mapStateToProps` is used to map the `counter` value from the store to the `value` prop of a component:

```
import { connect } from 'react-redux';
```

```
const mapStateToProps = (state) => {
  return {
    value: state.counter
  }
}

const CounterDisplay = ({ value }) => (
  <div>
    {value}
  </div>
);

export default connect(mapStateToProps)(CounterDisplay);
```

In this example, the `mapStateToProps` function is used to map the `counter` value from the store to the `value` prop of the `CounterDisplay` component. The component can then access and display the current value of the counter.

It's important to notice that `mapStateToProps` is not necessary if the component does not need to access any values from the store, in that case, `connect` function can be called without passing this function.

What is react-redux ?

`react-redux` is a library that allows React and Redux to work together. It provides a way to connect React components to the Redux store, allowing the components to access and update the state of the store.

`react-redux` provides a higher-order component called `connect` that allows you to connect your React components to the Redux store. It also provides a `Provider` component that makes the store available to all components in the application.

Here is an example of how `react-redux` is used to connect a React component to the Redux store:

```
import { connect } from 'react-redux';

const mapStateToProps = (state) => {
  return {
    value: state.counter
```

```
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onIncrement: () => dispatch(incrementCounter())
  }
}

const CounterDisplay = ({ value, onIncrement }) => (
  <div>
    <div>{value}</div>
    <button onClick={onIncrement}>Increment</button>
  </div>
);

export default connect(mapStateToProps, mapDispatchToProps)(CounterDisplay);
```

In this example, the `connect` function is used to connect the `CounterDisplay` component to the Redux store. The `mapStateToProps` function is used to map the state of the store to the props of the component, and the `mapDispatchToProps` function is used to map the `dispatch` function to the props of the component. The component can then access the current value of the counter and dispatch actions to update the state of the store.

How is Redux different react-redux ?

Redux and react-redux are two different libraries that work together to manage the state of a React application.

Redux is a library that provides a way to manage the state of an application in a centralized, predictable way. It uses a store to hold the state, actions to update the state, and reducers to handle the actions. The state is updated by dispatching actions and the reducers handle the actions and update the state.

react-redux is a library that connects React components to the Redux store. It provides a way to map the state from the store to the props of the components, and map the actions to the props of the components so that they can dispatch the actions. It also provides the `<Provider>` component, which wraps the root component of the application and provides the store to all the child components.

In summary, Redux is the library that manages the state of the application and react-redux is the library that connects React components to the Redux store.

With the help of react-redux, React components can access the state of the store and dispatch actions to update the state.

A simple Redux Project

```
Folder structure

- src
  - store
    - index.js
    - reducers.js
    - actions.js
  - App.js
  - index.js
  - TodoList
```

1. Install the required libraries: `npm install redux react-redux`
2. Create a new folder called store and within it, create the following files:
    - actions.js: This file will contain the actions that will be dispatched to update the state.
    - reducers.js: This file will contain the reducers that will handle the actions and update the state.
    - index.js: This file will combine the actions and reducers and create the store.
3. In the actions.js file, create an action creator function that returns an action object. For example:

```
export const addTodo = (text) => {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

4. In the reducers.js file, create a reducer function that handles the actions and updates the state. For example:

```
const initialState = { todos: [] }

export const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return { ...state, todos: [...state.todos, action.text] }
    default:
      return state
```

```
  }
}
```

5. In the index.js file, import the createStore method from the redux library, the todoReducer from the reducers file and combine them.

```
import { createStore } from 'redux'
import { todoReducer } from './reducers'

const store = createStore(todoReducer)
export default store
```

6. In the root component of your application, import the Provider component from the react-redux library and wrap your root component with it. Pass the store as a prop to the Provider.

```
import { Provider } from 'react-redux'
import store from './store'

function App() {
  return (
    <Provider store={store}>
      <div className="App">
        {/* your components go here */}
      </div>
    </Provider>
  );
}
```

7. In the components that need to access the state and/or dispatch actions, import the useSelector and useDispatch hooks from the react-redux library. Use the useSelector hook to map the state to the component's props and the useDispatch hook to dispatch actions.

```
import { useSelector, useDispatch } from 'react-redux'
import { addTodo } from './store/actions'

function TodoList() {
  const todos = useSelector(state => state.todos)
  const dispatch = useDispatch()

  const handleAddTodo = (text) => {
    dispatch(addTodo(text))
  }

  const handleSubmit = (event) => {
```

```
    event.preventDefault();
    const formData = new FormData(event.target);
    const todoText = formData.get('todoText');

    handleAddTodo(todoText);
  }

  return (
    <div>
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>{todo}</li>
        ))}
      </ul>
        <form onSubmit={handleSubmit}>
          <input type="text" name="todoText" />
          <button type="submit">Add Todo</button>
        </form>
    </div>
    )
}
```

Session 9

Middleware

In the context of Redux, middleware refers to a way of handling and modifying actions before they reach the reducer. It allows you to add additional functionality to your application, such as logging, error handling, or API calls, by intercepting actions and performing certain tasks before they reach the reducer. This is useful because it allows you to separate the logic of your application into different middleware functions, making your code more modular and easier to manage. Or in other words, a middleware in Redux is a function that sits between the dispatch of an action and the moment it reaches the reducer. It allows us to modify, inspect or even stop the action from reaching the reducer. This can be useful for logging, handling async operations, or handling side effects.

Here is an example of a custom middleware that logs the action and state before and after the action is dispatched:

```
const loggerMiddleware = store => next => action => {
  console.log("Before action:", store.getState());
  console.log("Action:", action);
```

```
  next(action);
  console.log("After action:", store.getState());
};

const store = createStore(reducer, applyMiddleware(loggerMiddleware));
```

In the above example, `loggerMiddleware`

is a middleware that takes the store and the next middleware in the chain as arguments. It logs the state before and after the action is dispatched.

To use this middleware, it needs to be passed to the `createStore` function via the `applyMiddleware` utility.

We can also use third party middleware libraries like `redux-thunk`, `redux-saga`, `redux-observable` to handle async operations and side effects in a more organized and efficient way.

Redux Thunk

Redux Thunk is a middleware that allows you to write action creators that return a function instead of an action. This function can then dispatch multiple actions, handle async logic, or anything else you need before the actual action is dispatched. It helps to handle async logic such as fetching data from an API, handling a form submission, or performing a complex calculation before dispatching an action. This makes it easy to handle complex logic and maintain a clean and organized codebase.

Here is an example of a simple action creator that uses a thunk to make an API call and dispatch a success or error action based on the response:

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

export const fetchData = () => async (dispatch) => {
  try {
    const response = await
axios.get('https://jsonplaceholder.typicode.com/todos/1');
    dispatch({ type: 'FETCH_DATA_SUCCESS', payload: response.data });
  } catch (error) {
    dispatch({ type: 'FETCH_DATA_ERROR', payload: error });
  }
```

```
};

// component.js
import { useEffect } from 'react';
import { useDispatch } from 'react-redux';
import { fetchData } from './actions';

export const MyComponent = () => {
  const dispatch = useDispatch();

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

  return <div>Loading...</div>;
};
```

In the example above, `fetchData` is an action creator that returns a thunk function. The thunk function is invoked by the `dispatch` function and can access the `dispatch` function as well as the current state of the store via its arguments. In this case, the thunk makes an API call and dispatches a success or error action based on the response.

In order to make the thunk middleware work, you will need to apply it to the store while creating it.

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));
```

This makes the `dispatch` function to also support functions (thunks) as actions.

This allows for much more flexibility and control over the flow of your application's state, as well as handling of async logic, such as API calls.

Combining Reducers

In Redux, multiple reducers can be combined using the `combineReducers` function from the `redux` library. This function takes an object as an argument, where the keys of the object represent the state keys and the values represent the

corresponding reducers. The resulting combined reducer will handle actions dispatched to any of the individual reducers, updating the corresponding state key.

Here is an example of combining two reducers :

```
import { createStore, combineReducers } from 'redux';

const reducerA = (state = {}, action) => {
  switch (action.type) {
    case 'ACTION_A':
      return { ...state, data: action.payload };
    default:
      return state;
  }
};

const reducerB = (state = {}, action) => {
  switch (action.type) {
    case 'ACTION_B':
      return { ...state, data: action.payload };
    default:
      return state;
  }
};

const rootReducer = combineReducers({
  reducerA,
  reducerB
});

const store = createStore(rootReducer);

console.log(store.getState().reducerA); // { data: undefined }
console.log(store.getState().reducerB); // { data: undefined }

store.dispatch({ type: 'ACTION_A', payload: 'Hello' });
console.log(store.getState().reducerA); // { data: 'Hello' }
console.log(store.getState().reducerB); // { data: undefined }

store.dispatch({ type: 'ACTION_B', payload: 'World' });
console.log(store.getState().reducerA); // { data: 'Hello' }
console.log(store.getState().reducerB); // { data: 'World' }
```

To access the store after combining multiple reducers in Redux, you can use the `getState()` method provided by the store. This method returns the current state of the store, which includes the state managed by all the combined

reducers. For example, if you have two reducers, `reducerA` and `reducerB`, and they are combined using the `combineReducers()` method, you can access the state managed by `reducerA` using `store.getState().reducerA` and the state managed by `reducerB` using `store.getState().reducerB`. In above example, `rootReducer` is combination of two reducers `reducerA` and `reducerB` and we are able to access the state of each reducer using the store object.

To run the above code, you would first need to have a development environment set up with Node.js and a package manager such as npm or yarn. Once that is done, you can create a new project directory and initialize it as a Node.js project by running "npm init" or "yarn init" in the terminal. Then, you can install the necessary dependencies such as Redux and any other libraries used in the code.

After that, you can create the necessary files such as the store, actions, and reducers and fill them with the code provided. Finally, you can start the project by running a command such as "npm start" or "yarn start" that starts a development server and runs the project in the browser.

It's worth noting that the example provided is likely incomplete and may not run as is, and would need additional work to make it run.

Redux hooks

Redux provides several hooks that can be used to manage the state in functional components. These hooks include:

1. useDispatch: This hook allows you to access the store's dispatch function. This can be used to dispatch actions to the store. Example:

```
const dispatch = useDispatch();

function handleClick() {
    dispatch(someAction());
}
```

2. useStore: This hook allows you to access the entire store object. This can be useful in advanced cases where you need to access the store directly. Example:

```
const store = useStore();
```

```
console.log(store.getState());
```

react-redux hooks

React-Redux has several hooks that are not provided by Redux, these hooks are used to connect your React components to the Redux store. The most commonly used hooks are:

1. useSelector: This hook allows you to access the state of your Redux store in your functional components. It takes a selector function as an argument and returns the selected state. The second parameter of the `useSelector` hook is an optional equality comparison function. By default, the hook uses a shallow equality comparison (Object.is) to determine whether the selected state has changed. However, you can pass in a custom function as the second parameter to use a different comparison method.

```
const myState = useSelector(
  state => state.myObject.specificProperty,
  (prev, next) => prev.id === next.id
);
```

   In this example, the custom function compares the `id` property of the previous and next values of the `specificProperty` state, and returns `true` if they match, indicating that the state has not changed and `useSelector` should not re-render the component. In the code example provided, `prev` and `next` refer to the previous and next states respectively, as they are being passed as the second parameter to the `useSelector` hook. This allows you to compare the previous state to the current state and make decisions based on the changes. This is useful for scenarios where you want to do something when a certain condition becomes true or false, for example you may only want to update a component when a certain value in the state changes.
2. useDispatch: This hook allows you to dispatch actions to the Redux store from your functional components. It returns the dispatch function from the store.

```
import { useDispatch } from 'react-redux'

function Component() {
  const dispatch = useDispatch()
  const handleClick = () => dispatch(someAction())
```

```
  return <button onClick={handleClick}>Dispatch</button>
}
```

3. useStore: The `useStore` hook allows you to access the Redux store directly in your functional component. It takes no parameters and returns the store object, which you can then use to dispatch actions or get the current state. Here is an example of how to use `useStore`:

```
import { useStore } from 'react-redux'

function MyComponent() {
  const store = useStore()
  const state = store.getState()
  const dispatch = store.dispatch

  // use state and dispatch as needed
  ...
}
```

In the above example, `useStore` is imported from `react-redux` and then called within the component to get the store object. The `getState` method is used to get the current state, and the `dispatch` method is used to dispatch actions.

It is important to note that `useStore` is not recommended to be used in most cases, as it bypasses the performance optimizations provided by the `useSelector` and `useDispatch` hooks. It is mainly used for advanced use cases where direct store access is needed.

These hooks are not provided by Redux, they are specific to React-Redux library and are used to connect your React components with the Redux store.

Session 10

Building an e-commerce application (Link)