Hyperiondev

TASK

React Hooks

Visit our website

Introduction

WELCOME TO THE REACT HOOKS TASK!

Originally, React mainly used class components, which can be a hassle as you always had to switch between classes, higher-order components, and render props. You can now do all these without switching, using function components with React hooks.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

WHAT ARE REACT HOOKS?

Before React 16.8.0, the most common way of handling lifecycle events required ES6 class-based components. For instance, if the code was written using React function components, you needed to rewrite those components as classes that extend React Component with a dedicated render function. Only after that would it be possible to access the lifecycle methods. With React Hooks, it becomes possible to use state and other features in a function component without the need to write a class or define the render method.

Hooks are JavaScript functions that manage the state's behaviour and side effects by isolating them from a component.

State Management

CLASS COMPONENTS

Class components extend from the React.Component class. Component objects have a state, meaning the object can contain information or data about the component that can change over the object's lifetime. Class components respond to lifecycle methods such as ComponentDidMount( ), ComponentDidUpdate( ), and ComponentWillUnMount( ).

Lifecycle methods enable updated state information to trigger a re-render, which updates the DOM with revised HTML markup.

```
class DisplayCount extends React.Component  {
  constructor(props) {
   super(props);
      this.state = {
          count: 0
      };
  }
  componentDidMount() {
      this.setState({
          count: this.props.count
      })
  }
  render() {
      return <h1>Count : {this.state.count}</h1>;
  }
}
```

In the example, mouse clicks increment the count, which is then displayed.

FUNCTION COMPONENTS

Function components are JavaScript functions that use React hooks to provide the equivalent functionality as class components. We call them components because the functions are constructed with single props object arguments and return React elements. Although function components have the word "component" in their name, they don't extend from the React component class.

Since function components are not objects; you must use React hooks to manage state and lifecycle events.

Function components are more concise, leading to cleaner, less complex code. They don't include lifecycle methods or inherited members required for code functionality.

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

  useEffect(() => {
      setCount(props.count);
  }, [props.count] );

  return <h1>Count : {count}</h1>;
}
```

In the example above, we revisit the code that displays the count. This time we've implemented it as a function component using hooks.

Notice how accessing the state using the "this" keyword is no longer necessary.

The useState hook

DECLARING STATE VARIABLES

The useState declares a state variable. It is the alternative to *this.state* used in class components. State variables are preserved between function calls.

The hook accepts the initial state of the variable as its argument and returns a pair of values, the current state, and a function that updates it.

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

   useEffect(() => {
       setCount(props.count);
   }, [props.count] );

   return <h1>Count : {count}</h1>;
}
```

In the above example, we declare a state variable called count and set it to zero. It is also possible to declare multiple state variables of different data types.

```
  const [clicks, setClick] = useState(0);
const [age, setAge] = useState(27);
const [colour, setColour] = useState('orange');
const [bookList, setBookList] = useState([{ text : 'The Chronicles of Narnia'}])
```

However, when there are multiple values that you need to keep track of, it would be better to declare a single object state as opposed to numerous state variables. A good use case example would be a form with many inputs that are saved/updated through a single API.

```
   const [user, setUser] = useState( {name: 'Robert', age: 27})
```

UPDATING STATE

In a class component, we call this.setState( ) to update state variables . Using hooks, we use the update function passed into useState( ).

State variables should be immutable. Instead of the state updating, the set function will replace the variable value. Without using the state setting function, React has no idea that the object has changed, so a re-render may not be triggered, which would affect the UI.

```
function DisplayClickCount(props) {

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

```
  return (
      <div>
          <h1>Clicks : {clicksCount}</h1>
          <button onClick = {() => setCount(clicksCount + 1)}>
              Click the button
          </button>
      </div>
  );
}
```

The useEffect hook

The useEffect hook helps manage function components' side effects; that is, any function you need to run after updating the DOM. It replaces some events by running a function whenever one or more variables change. It takes two arguments: a function and an optional array. The function defines which effect to run, and the optional array indicates variables, objects, etc., to watch for changes.

It serves the same purpose as ComponentDidMount( ), ComponentDidUpdate( ), and ComponentWillUnMount( ) in class components but bound into a single API.

The first argument of the hook is a callback function called *effect* that either returns a cleanup function or is undefined. The effect is executed when the component is mounted, and whether it is invoked in subsequent updates is determined by the dependency array passed as the second argument (optional).
The dependency array instructs the hook to run the callback only when there is a change to any of the elements in the array.

```
class ClassComponent extends React.Component {
   componentDidMount() {
   console.log("Hello");
   }
   render() {
   return <h1>Hello World</h1>;
   }
}

const functionComponent = () => {
   React.useEffect(() => {
     console.log("Hello");
```

```
  }, []);
  return <h1>Hello World</h1>;
};
```

In the above example, we have replaced componentDidMount with the useEffect hook and an empty dependency array as its second argument meaning that it is only invoked once on mounting.

By default, effects run after every completed render, but you can choose to invoke them only when specific values have changed.

```
const [title, setTitle] = useState('Default');

  useEffect(() => {
      document.title = title;
  }, [title]);
```

In the above example, the hook is invoked when the variable "title" is updated.

FETCHING DATA FROM API

In the code snippet below, we demonstrate how to fetch a random image using useEffect and the JavaScript fetch API to make an asynchronous HTTP request.

The default value of the randomImage variable is set to null so that the image is only rendered once the callback function has been executed successfully and the randomImage is updated with the URL response.

```
let [randomImage, setRandomImage] = useState(null)

useEffect(() => {
  fetch("https://random.imagecdn.app/500/150")
  .then(response => response.json())
  .then(data => setRandomImage(data.message))
},[])
```

UNSUBSCRIBING FROM LISTENERS

The useEffect hook uses resources such as a subscription/timer that may need to be terminated once its purpose has been fulfilled. If this isn't handled, the code may attempt to update a state variable that no longer exists, resulting in a memory leak. To avoid this, we implement an effect cleanup function within the useEffect hook when the component is unmounted.

Cleanup is only required when we need to terminate a repeated effect when a component unmounts.

```
const [time, setTime] = useState(0);

useEffect(() => {
  let interval = setInterval(() => setTime(1), 1000);
  return () => {
    clearInterval(interval);
  }
}, []);
```

In the snippet above, setInterval is terminated once the component unmounts.

The useRef hook

This hook lets you directly reference the function component's DOM and store mutable values that won't trigger a re-render when updated. We can use the hook to track state changes with the useEffect and useState hooks.

In the code snippet below, we use the useRef hook to keep track of the application renders.

```
function countRender() {
  const [inputValue, setInputValue] = useState("");
  const count = useRef(0);
  useEffect(() => {
    count.current = count.current + 1;
  });
  return (
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <h1>Render Count: {count.current}</h1>
    </>
  );
}
```

The useContext hook

This hook helps build a context API, a mechanism used to share data without passing props. It returns the current value for a context.

It makes up part of React's Context API along with the Provider and Consumer components.

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

function App() {
 return (
   <NumberContext.Provider value={27}>
       <div>
          <Display />
       </div>
   </NumberContext.Provider>
 );
}

function Display() {
   const value = useContext(NumberContext);
   return
       <div>

          The answer is {value}.

       </div>;
}

ReactDOM.render(<App />, document.querySelector("#root"));
```

In the example above, we create a new context object, NumberContext, with Provider and Consumer properties. The Provider is rendered and passed the value

In the example above, we create a new context object, NumberContext, with Provider and Consumer properties. The Provider is rendered and passed the value prop, which can be accessed using either the Consumer or useContext as we have done in this case. We pass the context object into the useContext hook to read the value.

The useReducer hook

This hook stores the current state value and is an alternative to the useState hook. The hook helps in separating render and state management but extracting the state management from the component.

The hook accepts two arguments - the *reducer* function and the initial state. It returns the current state and dispatch function.

Let's break down what all of that means.

The dispatch function is initialised through the useReducer hook. It dispatches an action object. When you want to update a state, you call the dispatch function with the appropriate action object, which describes how to update the state.

The reducer function accepts two arguments; the current state and an action object. It updates and returns the new state value.

The example provided is of a stopwatch implementation. The initial state sets the time to zero. The event handlers of the buttons use the dispatch function to dispatch the appropriate action object. Each time the reducer function updates the state, the component is re-rendered and receives a new state.

```
function Stopwatch() {
 const [state, dispatch] = useReducer(reducer, initialState);
 const idRef = useRef(0);
 useEffect(() => {
   if (!state.isRunning) {
     return;
   }
   idRef.current = setInterval(() => dispatch({type: 'tick'}), 1000);
   return () => {
     clearInterval(idRef.current);
     idRef.current = 0;
   };
 }, [state.isRunning]);
  return (
   <div>
     {state.time}s
     <button onClick={() => dispatch({ type: 'start' })}>
       Start
     </button>
     <button onClick={() => dispatch({ type: 'stop' })}>
       Stop
     </button>
     <button onClick={() => dispatch({ type: 'reset' })}>
       Reset
     </button>
   </div>
 );
}
```

Compulsory Task

- Modify your solutions to the compulsory tasks for State Management and Component Lifecycle to only use function components and hooks.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.