Hyperiondev

TASK

React - Local State Management and Events

Visit our website

Introduction

WELCOME TO THE REACT - LOCAL STATE MANAGEMENT AND EVENTS TASK!

Your React apps will become more useful with each task. You are now able to add attractive components to your UI. However, for your React apps to be truly responsive, your components need to be able to react to events. We will discuss how to do this in this task. We will also learn how to use React-Router to render and display different web pages.

REACT EVENTS

The concept of event-driven programming should be familiar to you. We have already created several web applications that have responded to events. Unsurprisingly, React applications are event-driven as well. In this section, you will learn how to handle events with React.

React supports a host of events. You can find a list of these events **here**.

To create a component that responds to a specific event you must:

1. Create a functional component (as shown in the previous task).
2. Create an event handler that responds to the event.
3. Register the React component with the event handler.

Consider the following **example** which illustrates how this is done:

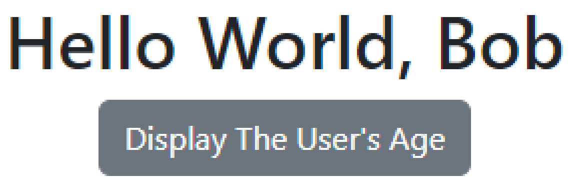
Welcome.js

```
// Importing react
import React from "react";

function Welcome(props) {
  // An event handler function that displays a message in the console
    const displayAge = () => {
    console.log(props.age);
  };

  return (
    <div>
      {/* Display the name value of the props object */}
      <h1>Hello World, {props.name}</h1>
      {/* This button will call the displayAge function when clicked */}
      <button onClick={displayAge}>Display The User's Age</button>

    </div>
  );
}
export default Welcome;
```

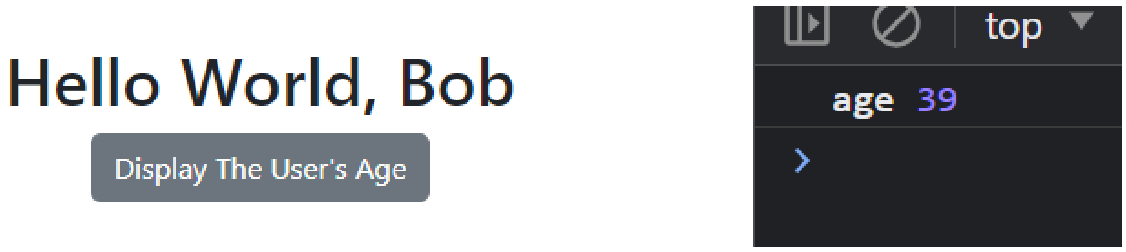
Now we import the Welcome component to the **App.js** file. To test it out:

```
// Importing components and CSS files
import Welcome from './Components/Welcome';
import './App.css';

// Create the App component
function App() {
  return (
    <div className="App">
      {/* Display an instance of the Welcome component
      passing 2 props
      name="Bob"
      age="39" */}
      <Welcome name="Bob" age="39" />
    </div>
  );
}

export default App;
```

When we run the application in the browser, the **Welcome** component is rendered and displays the heading and button tags. The heading will include the `name` prop, and the button will execute the console.log method in the browser's console, displaying the `age` prop.



And the log displayed when we click on the button:



```
age 39
```

Here are some important points to note regarding handling events with React:

- Use camelCase notation to name React events.
- As shown in the example above, with JSX you must pass a function (in the curly braces **{}**) as an event handler. **Do not** actually call the function here — notice the lack of parentheses after the function name.

```
    <button onClick={displayAge}>Display The User's Age</button>
```

You can have as many event handlers in a component as you need:

```
import React from "react";

function Welcome(props) {
// An event handler function that displays a console.log message
  const displayAge = () => {
    console.log(props.age);
  };
// An event handler function that changes the background color of the web page
  function changeBackgroundColor() {
    let bodyStyle = document.body.style;
    if (bodyStyle.backgroundColor === "black") {
      bodyStyle.backgroundColor = "white";
    } else {
      bodyStyle.backgroundColor = "black";
    }
  }

  return (
    <div>
      <h1>Hello World, {props.name}</h1>
      {/* This button will call the displayAge function when clicked */}
      <button onClick={displayAge}>Display The User's Age</button>
         {/*  This  button  will  call  the  changeBackgroundColor  function  when
clicked */}
          <button  onClick={changeBackgroundColor}>  Change  Background  Color
</button>
    </div>
  );
}
export default Welcome;
```

REACT STATE

What is state?

The state is a built-in React object that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders.

What is the difference between State and Props in a React Component?

State

```
  //state variable and setter function. State value is 0.
  const [count, setCount] = useState(0)
```

The **state** is used to store and manage data within a component. When a component's state changes, React automatically re-renders the component, reflecting the updated state in the UI.

Props

```
{ /* number & btnClick is the prop name and count is the state value */ }
  <ChildOne number={count} btnClick={callback} />
```

Props: short for properties, are used to pass data from a parent component to its child components. This allows child components to receive and use data from their parent, making it easy to share data between components, but they cannot directly change the parent's state.

You can pass any type of data or even functions as props to children's components, which can then use that data or invoke the functions. The data you want to pass to a component are defined inside the curly braces.

Props are immutable and cannot be changed within a component, while the state is mutable and can be updated using the `setCount` function inside the parent component. Even though it's possible to pass the `setCount` as props, it is not recommended as it will cause unexpected behaviour as your code grows. It's recommended to use a callback function instead:

```
 function handleState(){
   setCount(count + 1)
 }
 // pass the function as props to be invoked in the child component
<ChildOne updateState={handleState}/>
```

Let's look at a simple example:

First we will write a React component called **Count.js** that will keep track of the number of times the user has clicked a button.

The **Count.js** component contains an <h1> element that displays the value of a variable **{count}**, and a <button> that has an **onClick** event that calls a function **{increaseCount}** which increments the value of the count variable each time it is clicked:

Count.js

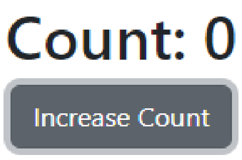
```
import React from 'react';
export default function Count() {
    // Event handler for the onClick event
    function increaseCount() {
        count += count;
        console.log("Event Triggered");
      }
   // Create a variable called "count" and set the initial value to 0
    let count = 0
  return (
    <div>
      {/* use JSX code to display the value of the count variable */}
      <h1>Count : {count}</h1>
      {/* Button to increase the value of the count variable */}
      <button onClick={increaseCount} > Increase Count </button>
    </div>
  )
}
```

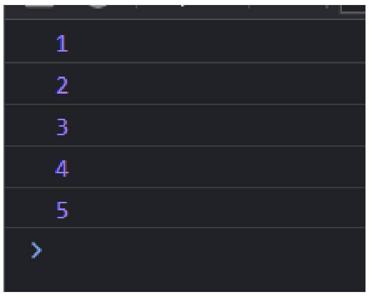
Now, we import the **Count.js** component into our **App.js** file:

App.js

```
// Import components and files
import "./App.css";
import Count from "./Components/Count";

// Create the App component
function App() {
  return (
    <div className="App">
      {/* Import the count component */}
      <Count />
    </div>
  );
}

export default App;
```

Run the Application and open it in your browser. This should appear in your browser:





```
1
2
3
4
5
>
```

When testing the application by clicking on the "Increase Count" button, the value displayed will not change.

However, the **console.log** in the event handler will show the value incrementing. But the value on the UI stays the same. This is because React re-renders the component whenever the state or props change but does not re-render when another event takes place or variables are changed, like in this instance a button is clicked. This is why we need **State** in a React component: to “remember” any changes that have happened in the component and trigger a re-render that will display the changes.

Let’s consider how to add state to a component using React State Hooks. The first thing you might ask yourself is: what React Hooks are. The answer is that React hooks are powerful functions that can be used in a functional component to manipulate the state of the component.

Let's add some state to the component to **keep track** of how many times the user has clicked the button.

First, import **useState** from React:

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

Now you can declare a state variable inside your component:

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

You’ll get two things from useState: the current state (count), and the function that allows you to update it (setCount). You may give them any names, but the convention is to write [something, setSomething].

And change the function to use the **setCount** hook to change the value of the **count** state variable:

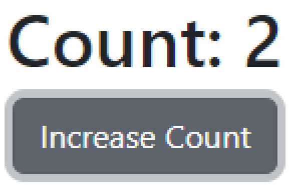
```
    function increaseCount() {
        setCount(count + 1);
      }
```

After the changes, our **Count.js** component should look like this:

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

export default function Count() {
    // create a state variable count with an initial value of 0
    const [count, setCount] = useState(0);

    // Event handler for the button changes the value of the count state
    function increaseCount() {
        setCount(count + 1);
      }

  return (
    <div>
      {/* use JSX code to display the value of the count variable */}
      <h1>Count : {count}</h1>
      {/* Button to increase the value of the count variable */}
      <button onClick={increaseCount} > Increase Count </button>
    </div>
  )
}
```

When we run the application, open it in our browser, and click the button. The value of Count will increase each time the button is clicked. State allows the component to "remember" the value.

The first time the button is displayed, count will be 0 because you passed 0 to useState(), setting the initial value to 0. When you want to change state, call setCount() and pass the new value to it. Clicking this button will increment the counter:



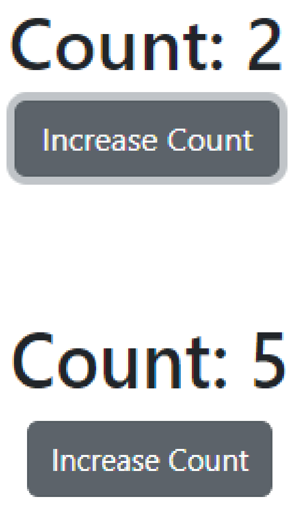
Now we can create a component, and we can have it remember the value of variables using state. Each instance of a component will have its own unique state.

Let's test this out by adding another instance of the Count component to **App.js**:

```
// Import components and files
import "./App.css";
import Count from "./Components/Count";
```

```
// Create the App component
function App() {
  return (
    <div className="App">
      {/* Create an instance of the count component */}
      <Count />
      <hr />
      {/* Create  a second instance of the count component */}
      <Count />
    </div>
  );
}

export default App;
```

If we run the application and open it in our browser the result will look something like this:



As you can see, the state for each of the Count components is unique and will not affect any other instances of the same component.

REACT INPUT

Next, we look at getting **input** from the user.

Earlier in the task we created a **Welcome.js** component that displays a name using props. Now we will use state and an input element to change the values that are displayed.

First we create the **NameInput.js** component that has a state variable called name in the components folder:

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

export default function NameInput() {
    // Create a name state object with a default value
  const [userName, setUserName] = useState("");

  return (
    <div>
      <label for='nameInput' >Enter Name: </label>
        <input
      {/* Input component using onChange to update the value of the state */}
          onChange={(event) => setUserName(event.target.value)}
          id="nameInput"
          placeholder="Enter name here"
          defaultValue={userName}
        />
      {/* Display the value of the userName State variable */}
      <h3> Name : {userName}</h3>
    </div>
  );
}
```

We are using several properties in this input element:

- **id="nameInput"** - Is here specifically used to link the input to the label.
- **defaultValue={userName}** - This property sets the initial value displayed in the input element.
- **<h3>...{userName}</h3>** - this refers to the value typed into the input element, by setting the value to the **{userName}** state variable, we ensure that the value of the state object will always match the value the user has entered.

- **onChange** - We set the **onChange** prop on the field, so every time its value changes, the inline anonymous (arrow) function is invoked updating the state.

onChange is an Event handler function required for controlled inputs. The onChange event fires immediately when the input's value is changed by the user (for example, it fires on every keystroke). It behaves like the browser input event.

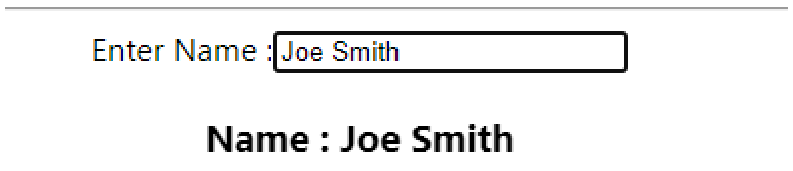
We can access the value of the input element as **event.target.value** in the anonymous (arrow) function as the **event** object is passed as a parameter to the function. The **target** property on the **event** object refers to the input element that triggered the **event**.

```
onChange={(event) => setUserName(event.target.value)}
```

Next, as always, we import the component to **App.js**:

```
// Import files and components
import './App.css';
import NameInput from './Components/NameInput.js'

// Create the App component
function App() {
  return (
    <div className="App">
      <hr />
      {/* Create an instance of the NameInput component */}
      <NameInput />
    </div>
  );
}

export default App;
```

Now when we run the application and open it in our browser we should see something similar to this:


Enter Name : Joe Smith

Name : Joe Smith

The **{userName}** value displayed in the <h3> element:

```
<h3> Name : {userName}</h3>
```

This now matches any input the user types into the <input> element.

We can now Get input from the user but we still need to pass that state to the Welcome.js component.

Sharing state between components

Often we want the state of two components to always change together. To do this, we will need to remove state from both of the components and move it to the closest parent component and then pass it to them via props. This is referred to as "lifting state up", and it's one of the most common things you will do while writing React code.

In our example, we want to share the state between the **NameInput.js** and **Welcome.js** components. The simplest way to do this would be to add our state to the **App.js** file. However, this is a bad idea because adding state to the **App.js** component can quickly become cumbersome and unmaintainable if the application grows larger than a few components. To keep the code as modular as possible, we will create a new parent component **DisplayName.js** that will have both **NameInput.js** and **Welcome.js** as its child components. To do this we will create the **DisplayName.js** component and import both the **NameInput.js** and **Welcome.js** components into it.

We will also need to lift the state management out of the **NameInput.js** into the **DisplayName.js** component.

First, we create a parent component called **DisplayName.js** In the components folder:

```
import React from "react";
// Import the Child components
```

```
import Welcome from "./Welcome";
import NameInput from "./NameInput";
// Import useState
import { useState } from "react";

export default function DisplayName() {
  // Create the name state variable
  const [userName, setUserName] = useState("");

  // Create a function to handle the state change
  function handleChange(userInput) {
    setUserName(userInput);
  }

  return (
    <div>
      {/* The userName state is passed to the Welcome component as a prop */}
      <Welcome name={userName} />
      {/* The userName state is passed to the NameInput component as a prop
          The handleChange function is passed to the component as a prop */}
      <NameInput name={userName} handleChange={handleChange} />
    </div>
  );
}
```

The **DisplayName.js** component creates the **userName** state which is passed to both child components as props and the **handleChange** function that is passed to the **NameInput.js** component as a prop:

```
<NameInput name={userName} handleChange={handleChange} />
```

Because we have lifted the state out of the **NameInput.js** component, the component no longer contains state. Both the **userName** state and the **handleChange** function are passed as props.

We can edit the component to use the props instead of its own state:

```
import React from "react";
```

```
export default function NameInput(props) {
  // The props passed to this component include the state variable 'name',
  // and the handleChange function that changes the state of name is passed
     as a prop

  return (
    <div>
      {/* Input component using onChange to update the value of the state */}
      <label>
        Enter Name :
        <input
          // The onChange event calls the handleChange function
             that was passed from the parent component as a prop
          onChange={(event) => props.handleChange(event.target.value)}
          name="nameInput"
          defaultValue=" Enter name here"
          value={props.name}
        />
      </label>

      {/* Display the value of the name State variable */}
      <h3>  Name :  {props.name}</h3>
    </div>
  );
}
```

The **Welcome.js** component stays largely unchanged, having merely removed the **age** prop and **thedisplayAge** function as they are not needed:

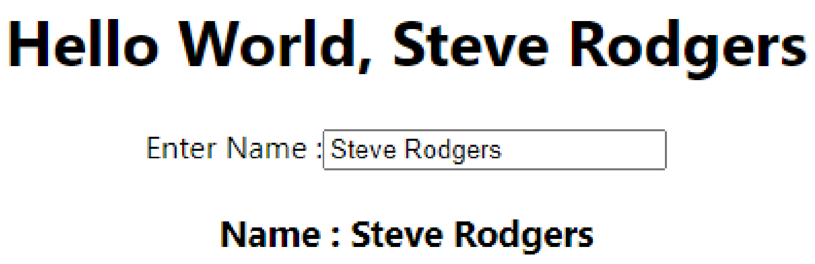
```
import React from "react";

function Welcome(props) {
  return (
    <div>
      {/* Display the name value of the props object */}
      <h1>Hello World, {props.name}</h1>
    </div>
  );
}
```

```
export default Welcome;
```

Next we edit the **App.js** file to import and display the **DisplayName.js** parent component and not the two child components.

```
// Import files and components
import "./App.css";
import DisplayName from "./Components/DisplayName";


// Create the App component
function App() {
  return (
    <div className="App">
      {/* Create an instance of the DisplayName component */}
      <DisplayName />
    </div>
  );
}

export default App;
```

When we run the application and open it in our browser the results should look similar to this:



```
Hello World, Steve Rodgers

Enter Name : Steve Rodgers

Name : Steve Rodgers
```

Any input that the user types into the input element in the **NameInput.js** component is lifted to the **DisplayName.js** parent component, which passes the same data as props to both child components.

Compulsory Task 1

Create an interest calculator web page with React and JSX with the following content:

- Create a folder called `ReactComponents` on your local machine.
- Open the command line interface/terminal and `cd` to the folder you have created above.
- Follow the instructions found **here** to install React using the Create React App (CRA) Starter Kit. Name your React App `react-interest-calculator`.
- Once you have started your front-end server (with `npm start`), test the default React App that was created by CRA by navigating to `http://localhost:3000/` in your browser.
- Create a website that will simulate a banking application with the following functionality:
    - The app should display the user's current bank balance.
    - The app must have an input for the user to deposit money to the bank (the user should input a number and click a button that will add the "deposit" amount to the currently displayed bank balance).
    - The app must have an input for the user to withdraw money from the bank (the user should input a number and click a button that will remove the withdrawn amount to the currently displayed bank balance).
    - There should be a button that the user can click to 'add interest' to the account using either a fixed interest rate % or a rate the user has entered in another input that will then be added to the balance being displayed.
    - There should be a button that the user can click to 'Charge bank fees' that can be either a fixed amount or calculated as a percentage of the bank balance that is then deducted from the displayed balance.
    - This app must use at least **two** separate components with a shared state that is lifted to a parent component.
- Try to be creative and use some Bootstrap styling, or **optionally**, have an alert that triggers when the user goes into a negative balance, for example.

- Once you are ready to have your code reviewed, **delete** the `node_modules` folder (**please note**: this folder typically contains hundreds of files which, if you're working directly from Dropbox, has the potential to **slow down Dropbox sync and possibly your computer**), compress your project folder, and add it to the relevant task folder in Dropbox.

Remember to submit everything **except** the **node_modules** directory.

Rate us

Share your thoughts

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

Think 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.

REFERENCE

React.Dev. (2022). Quick Start. Retrieved 31 July 2023, from **https://react.dev/learn**