HyperionDev

React – Local State Management and Events

Task

Visit our website

Introduction

Welcome to the React – Local State Management and Events 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. Shortly, you'll learn to capture user input, update the UI in real time, and maintain the application state. By the end, you'll understand how to manage local state, handle events, and implement navigation, making your apps attractive, functional, and user-friendly.

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 on this **webpage on responding to events**.

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

1. Create a functional component.
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:

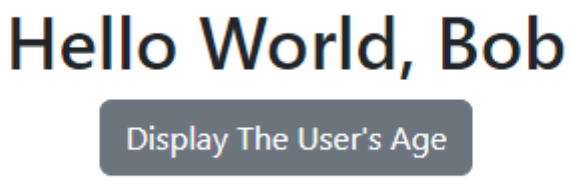
```
function Welcome({ name, age }) {
  // An event handler function that displays an alert message with the age
  const displayAge = () => {
    console.log(age);
  };

  return (
    <div>
      {/* Displaying the user's name passed as a prop */}
      <h1>Hello World, {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.jsx** 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. It should look like this:



The log is displayed when we click on the button, as shown below:

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

- Use camelCase notation to name React events.
- As shown in the example below, 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 also have as many event handlers in a component as you need. See below for an example of this:

```
function Welcome({ name, age }) {
  // An event handler function that displays a console.log message
  const displayAge = () => {
    console.log(age);
  };

  // An event handler function that changes the background colour of the web page
  const changeBackgroundColor = () => {
    // Retrieve the body element's style property
    let bodyStyle = document.body.style;

    // If the background colour is black, change it to white, otherwise change it
to black
    if (bodyStyle.backgroundColor === "black") {
      bodyStyle.backgroundColor = "white";
    } else {
      bodyStyle.backgroundColor = "black";
    }
  };

  return (
    <div>
      <h1>Hello World, {name}</h1>
      {/* This button will call the displayAge function when clicked */}
      <button onClick={displayAge}>Display The User's Age</button>
      {/* changeBackgroundColor is called when this button is clicked */}
```

```
      <button onClick={changeBackgroundColor}>Change Background Color</button>
    </div>
  );
}

export default Welcome;
```

React state

What is the state?

State is a built-in React object used to store data or information about a component. A component's state can change over time, and when it does the component automatically re-renders to reflect those changes.

What is the difference between state and props in a React component?

Example of state:

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

State is used to store and manage data within a component. When the state of a component changes, React automatically re-renders the component to reflect the updated state in the UI. State is local to the component and cannot be accessed or modified by child components directly.

Example of props:

```
{/* number & btnClick are props, and count is the state value being passed
down */}
<ChildOne number={count} btnClick={callback} />
```

Props, short for properties, are used to pass data from a parent component to its child components. They allow child components to receive data or functions from their parent, but child components cannot directly modify the parent's state. Props make it easy to share data between components, enhancing reusability.

You can pass any type of data or even functions as props to the 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, meaning they cannot be changed within a component once passed down from the parent. In contrast, state is mutable and can be updated within a component using the state setter function, such as `setCount`.

While you can pass a state setter function like `setCount` directly as props to a child component, this approach is not recommended as your application grows. It can lead to unexpected behavior, making the code harder to manage and debug.

Instead, it's better to use a callback function to update the parent's state:

```
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.jsx** that will keep track of the number of times the user has clicked a button.

The **Count.jsx** component contains an `<h1>` element that displays the value of a state variable called `count`, and a `<button>` with an `onClick` event that calls a function `increaseCount`. This function increments the `count` variable each time the button is clicked. See below for an example of this:

```
export default function Count() {
  // Event handler for the onClick event
  function increaseCount() {
    count += 1; // Increment the count variable
    console.log(count); // Log the updated count to the console
  }

  // 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.jsx** component into our **App.jsx** file:

```
// Import components and styles
import "./App.css";
import Count from "./components/Count"; // Import the Count component

// Create the App component
function App() {
  return (
    <div className="App">
      {/* Render the Count component to display and manage the count */}
      <Count />
    </div>
  );
}

export default App;
```

Run the application and open it in your browser. The following should appear in your browser:

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 (as in this instance where 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 see how to add state to a component using **React state hooks**. 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 a function that allows you to update it (`setCount`). You may give them any names, but the convention is to write them in the form `[something, setSomething]`.

Next, update the `increaseCount` function to use the `setCount` hook to change the value of the count state variable:

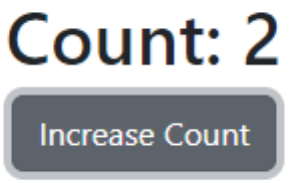
```
function increaseCount() {
    setCount(count + 1);
}
```

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

```
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>
  );
}
```

After updating **Count.jsx**, run the application, open it in the 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 zero because you passed zero to `useState()`, setting the initial value to zero. When you want to change state, call `setCount()` and pass the new value to it. Clicking this button will increment the counter:



Now when we create a component, 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.jsx**:

```
// 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, each `Count` component maintains its unique state, and changes in one instance do not affect any other instances of the same component.

React input

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

Earlier in the task, we created a **Welcome.jsx** 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 a **NameInput.jsx** component that has a state variable called `userName` in the components folder:

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

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

  return (
    <div>
      <label htmlFor="nameInput">Enter Name: </label>
      {/* Input component using onChange to update the value of the state */}
      <input
        onChange={(event) => setUserName(event.target.value)}
        id="nameInput"
        placeholder="Enter name here"
        value={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"` – This is specifically used to link the input to the label.
- `value={userName}` – This ensures that the input field is controlled, displaying the current value of `userName`.
- `<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.

The **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).

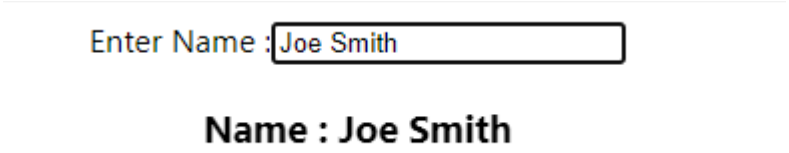
We can access the value of the input element as **event.target.value** in the anonymous (arrow) function, as the event object is passed as a parameter to the function. The target property on the event object refers to the input element that triggered the event.

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

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

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

// 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:



Name : Joe Smith

We can now get input from the user, but we still need to pass that state to the **Welcome.jsx** component.

Share 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.jsx** and **Welcome.jsx** components. The simplest way to do this would be to add our state to the **App.jsx** file. However, this is not ideal, because adding state to the **App.jsx** component can quickly become cumbersome and difficult to maintain 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.jsx**, that will have both **NameInput.jsx** and **Welcome.jsx** as its child components. To do this, we will create the **DisplayName.jsx** component, and import both the **NameInput.jsx** and **Welcome.jsx** components into it.

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

First, we create the parent component called **DisplayName.jsx** 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.jsx** component creates the `userName` state, which is passed to both child components as props, as well as the `handleChange` function that's passed to the **NameInput.jsx** component as a prop.

Because we have lifted the state out of the **NameInput.jsx** 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:

```
// Creating and exporting a functional component called NameInput
export default function NameInput({ name, handleChange }) {
  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) => handleChange(event.target.value)}
          name="nameInput"
          defaultValue=" Enter name here"
          value={name}
        />
      </label>

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

The **Welcome.jsx** component will stay largely unchanged, with only the `age` prop and the `displayAge` function removed as they are not needed at this stage:

```
// Create a functional component named Welcome
function Welcome({ name }) {
  return (
    <div>
```

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

// Exporting the created component.
export default Welcome;
```

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

```
// App.jsx
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:

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

Take note

The task below is **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give the task your best attempt and submit it when you are ready.

After you click “Request review” on your student dashboard, you will receive a 100% pass grade if you’ve submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task

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

- Open your command-line interface/terminal.
- Navigate to your desired location where the React application folder will be created using the `cd` command.
- Create a new React application called **react-interest-calculator** using the **React + Vite Starter Kit** by following the instructions found **here**, or you can use the command below:

```
npm create vite@latest react-interest-calculator -- --template
react
```

- Change into the newly created React application directory:

```
cd react-interest-calculator
```

- Install the necessary dependencies by running:

```
npm install
```

- Once your front-end server is running (`npm run dev`), open your browser and navigate to **http://localhost:5173/** to see the default React app created by Vite.

- Now, create a React application that simulates a banking system 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 from 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 percentage 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 either be a fixed amount or calculated as a percentage of the bank balance. This should then be 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.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

Share your thoughts

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

Do you think we’ve done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.