Hyperiondev

TASK

ReactJS II: Class Components and Props

Visit our website

Introduction

WELCOME TO THE REACT CLASS COMPONENTS AND PROPS TASK!

In the previous task, you learnt how to use JSX to create elements in React. However, React UIs are typically made up of components. I.e. the building blocks for all React applications. There are two main types of components: functional components and stateful components. In this task, you will learn about working with both these types of components.

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!

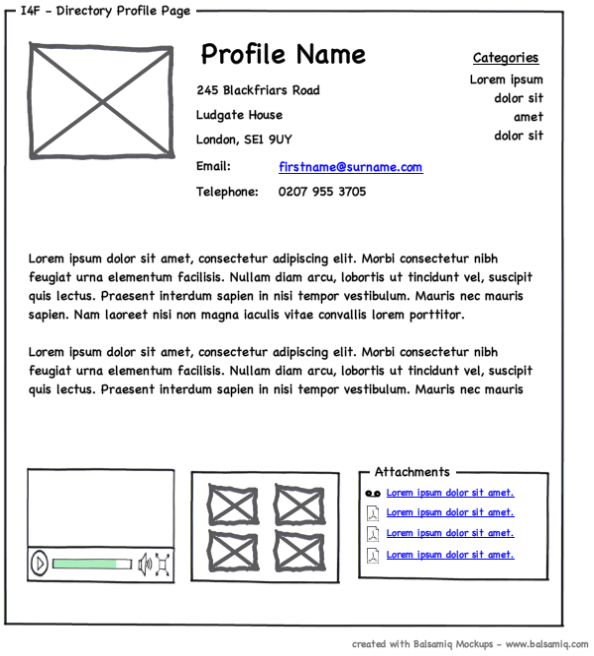
A note from the

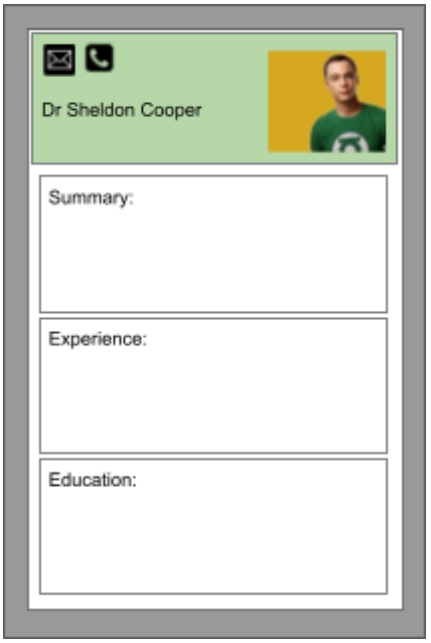
HyperionDev Team

In web development, technology moves at a rapid rate. Podcasts are an ideal way to stay up to date. They're an excellent fit for packed and busy lives because you can listen to them while on the move. For example, on **this site,** you can listen to weekly discussions with React developers. Read this Hyperion **blog post** for a list of other podcasts for web developers.

UI DESIGN

It is important to plan and design an app before you start coding. Before you even think about writing front-end code, you need to design the user interface (UI). There are various ways to do this including wireframing, which is a simple and effective technique of making sketches of how you would like your UI to look and work. Examples of wireframes are shown below:





All user interfaces (UIs) are made up of a number of components. This task is going to focus on the technical aspects of what a React component is and how to create

one. Before we consider that, however, let us discuss the logic of designing a UI in terms of components.

As you design your UI, you will notice that there are certain elements that will be used over and over again. For example, you might reuse menus, buttons and sections for comments etc.

According to **React's documentation**:

"A good rule of thumb is that if a part of your UI is used several times (Button, Panel, Avatar), or is complex enough on its own (App, FeedStory, Comment), it is a good candidate to be a reusable component."

A component can contain other components, e.g. you may have a button component that forms part of a header component.

Now that we understand what a component is, let's discuss how to implement them using React.

Extra resource

There is a real art to being able to create attractive and usable UIs! Unfortunately, a full discussion regarding UI design is beyond the scope of this Bootcamp but here are two good references to help you think about some of the most important principles involved in UI design:

- **The 4 golden rules of UI design**
- **User Interface design basics**

CREATING REACT COMPONENTS

Definition: *"React components are small, reusable pieces of code that return a React element to be rendered to the page."*

You learnt about React elements in the previous section. What is the difference between a React element and a React component? To understand this better, consider the example of a basic component in the code below:

```
function Welcome(props) {
```

```
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Sara" />;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
   {element}
  </React.StrictMode>
);
```

Notice that a React component is a piece of code (in this case a function) that accepts props as an input and returns a React element as output. React components can be implemented using functions (as in the example above) or using React classes (which we will consider later).

When creating React components it is extremely important that you always adhere to the following two rules:

1. **Start component names with a capital letter. React treats components starting with lowercase letters as DOM tags.**
2. **A component should NEVER modify its own props.**

Props definition: *"props are inputs to a React component. They are data passed down from a parent component to a child component."*

At this point, you probably have some questions. For example, you may be wondering what the point of React is if you aren't allowed to change anything. We've said previously that React elements are immutable (unchangeable) and now we are saying that React components are not allowed to change props. If we can't change anything with React we are going to end up with static web pages and we may as well use HTML, right? Not to worry - the solution lies in the fact that React components can store state information.

WHAT DATA SHOULD BE STORED IN STATE?

Let's briefly recap the definition of state. In React, *state is used to store data related to a component that changes over time*. This data affects how the component is rendered.

For example, let's look at an email app such as Gmail. This app is made up of many components and the appearance of these components changes as you interact with them. Consider the image below. The state of a checkbox can change from checked to unchecked as you interact with it. Similarly, a message can be read or unread. The component used to display the messages in the inbox will look different depending on the state of that component (i.e. whether the message is read or unread).

Using too much state will make a program more complex and unpredictable. Therefore, have as few components that use state (stateful components) as possible in your application. Save data using props instead of state wherever you can.

Use state when some data associated with a component changes over time. For example, you would need to store information about whether a checkbox is checked or not in its state. In other words, if a component needs to change one of its attributes over time, that attribute should be part of its state.

It is often the user that will change the state of a component by interacting with the UI.

```
function shouldIKeepSomethingInReactState() {
  if (canICalculateItFromProps()) {
    // Don't duplicate data from props in state.
    // Calculate what you can in render() method.
    return false;
  }
  if (!amIUsingItInRenderMethod()) {
    // Don't keep something in the state
    // if you don't use it for rendering.
    // For example, API subscriptions are
    // better off as custom private fields
    // or variables in external modules.
    return false;
  }
  // You can use React state for this!
  return true;
}
```

By Dan Abramov, creator of Redux

Only components that are defined using classes (not functional components) can have state. State is private to the class where it has been declared. No component knows whether another component has state or not, however, a component can choose to pass info stored in its state down to children using props.

In his book *React.js Essentials: A fast-paced guide to designing and building scalable and maintainable web apps with React.js*, Artemij Fedosejev highlights the following two principles that can help you decide when to use stateful or stateless components:

- "The minority of your React components are stateful. They should be at the top of your components' hierarchy. They encapsulate all of the interaction logic, manage the user interface state, and pass that state down the hierarchy to stateless components, using props."
- "The majority of your React components are stateless. They receive the state data from their parent components via `this.props` and render the data accordingly". (2015, Page 37).

Stateful components are being implemented using classes but stateless components are usually implemented using functions.

PROPS VS STATE

React has two ways of storing the data related to components: props and state.

Props	State
• Props can be used in functional and class components. • Props are used to pass data from parent elements to child elements. Props can only be passed from a parent to their children, not the other way around, i.e. a child cannot send props to its parent. This is known as top-down or unilateral data flow. • Props are read-only. They should not be modified by components.	• Only components defined as classes can have state, i.e. components implemented using functions instead of classes cannot have state. • State is private to its component. State is not accessible to any component other than the one that owns and sets it. Data about state can be sent as props.

	• A component can change its state (unlike props which cannot change).

HOW TO CREATE A STATEFUL COMPONENT

To create a stateful component, do the following:

Step 1: Create a class component.

A class component has the following structure:

```
class NameOfClass extends React.Component {
  render() {
    return (
      <div>
        <h1>Look at this example of a class component!</h1>
      </div>
    );
  }
}
```

Note that the class has a single render() method that returns the React Element.

For the rest of this section, we are going to be looking at an example of a stateful component created by React. We use the example of a component called 'Timer' that displays the number of seconds that a user has been viewing the page containing the component for. To see an example of this code in action, see the stateful component section on **this page**. The 'example' code for this task also includes this component and detailed comments that explain the code in more detail.

The code for this Timer component is given below. We will now analyse this code to see the steps involved in creating a stateful component. Notice that each step is colour-coded to match the corresponding code.

Step 2: Add a class constructor to assign the initial state.

Look at the code that is highlighted in mustard yellow in the code segment below to see how a constructor is created for a stateful component. The main function of the constructor is to set the initial state of the component. In the Timer component example, the initial state is set to 0 seconds.

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

  tick() {
    this.setState(prevState => ({
      seconds: prevState.seconds + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}

ReactDOM.render(<Timer />, mountNode);
```

Step 3: Read the state info (using **`this.state`**) and use it to render the component using the **`render()`** method.

Step 4: Allow modification of the state using **`this.setState()`**. **`this.setState()`** will usually be called in an event handler that handles a UI interaction. For example, you would call **`this.setState`** to change the state of a checkbox if it were checked or unchecked. Never modify the state directly (e.g. **`this.state.seconds = 15;`**) because this won't re-render the component. Always use **`this.setState()`**.

THE APP.JS COMPONENT

App.js is a component that is created by default when using Create React App. The App component is the container for all other React components. By default the

App component is rendered by calling `ReactDOM.render()` in the **index.js** file as shown below:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

registerServiceWorker();
```

Note: *App.js must start with a capital letter (i.e. App.js instead of app.js) because we want React to treat App as a component. Remember that all component names must start with a capital letter.*

DYNAMIC REACT COMPONENTS

By now you should have started to appreciate how state is used to create dynamic web applications with React. To make truly dynamic web applications, however, we need to be able to use React to do some logic. In this task, you will learn to control how and when components are displayed on the browser using if statements and lists.

Render components using if statements

One of the most basic things you want to be able to do with a React application is to decide which components to display, based on some conditions. For example, you may want to display different components in your header based on whether a person is logged into your site or not. You may want to display a welcome message if they are logged in or a button that allows them to sign in if they are not already signed in. Implementing this with React is simple.

Imagine you have already created the two components below:

```
function Welcome(props) {
  return <h1>Welcome back {props.name}!</h1>;
}

function SignUp() {
  return <button type="button">Sign in</button>;
}
```

You could use the logic below to decide which component to show:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  const userName = props.name;

  if (isLoggedIn) {
    return <Welcome name= {userName}/>;
  } else {
  return <SignUp />;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
   <Greeting isLoggedIn={false} name="John" />
  </React.StrictMode>
);
```

You could also manipulate an element inline using JSX. Remember with JSX you can embed any JavaScript using curly braces. For example:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

Render multiple components using lists

When designing UIs, you are also more than likely going to want to be able to add multiple instances of a component to your UI. For example, if you were creating a messaging app, you would need to render multiple messages to your UI. To do this using React, we make use of lists. We must be able to keep track of every component we have on the UI so it is important that we have a key (or unique identifier) for each item in the list.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
```

```
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
   <NumberList numbers={numbers} />
  </React.StrictMode>
);
```

If we analyse the code example shown above (taken from the **React documentation**) you will notice the following:

We create a functional component as usual and render it as usual.

To add multiple components, we also:

- Create an array (`numbers`). In most practical examples, this data would be read from a database or fetched from an API.
- Pass the array we created using props to the component we are creating (`numbers={numbers}`).
- Use the `map()` function to create a new array with the results of calling a function for every array element.
  This style of programming is called declarative, and is common in React. You can solve the same problem with a normal for-loop, which would then be imperative programming. For more info about the `map()` function see **here**.
- Give each item in the array a key that will uniquely identify that item (`key={number.toString()}`).

RECAP

In this task we have discussed many important concepts. To make sure you have understood all the key points, take note of the following:

- A React component is a small, reusable piece of code that returns a React element to be rendered to the page.
- A good rule of thumb is that, if a part of your UI is used several times (for example a button, panel or avatar), or is complex enough on its own (for example, a feedStory or comment), it is a good candidate to be a React component.
- React component names should always start with a capital letter, e.g. App.js

- You can create components using classes or functions.
- You can create stateless (components that do not contain state information) or stateful (components that contain state information and must be implemented using a class) components.
- Component data is stored in props or state. Components cannot change props but they can change their state.
- You can use if statements to decide which components to render and how.
- You can use lists to add multiple components.

Compulsory Task 1

Use React to create a website for a fictitious clothing brand.

- Create a header component that displays a logo (see **here** for help) and company name. It should be able to welcome a user to the site or ask them to sign in, depending on boolean input.

- Create a landing page component that tells the user about the company.

- Create a product component that displays information about the products that the company sells.

- Edit your existing **App.js** file so that it displays the header component, the landing page component and at least three product components. Use an array of values and props to do this. Pass props to the header component to specify whether the user is logged in or not.

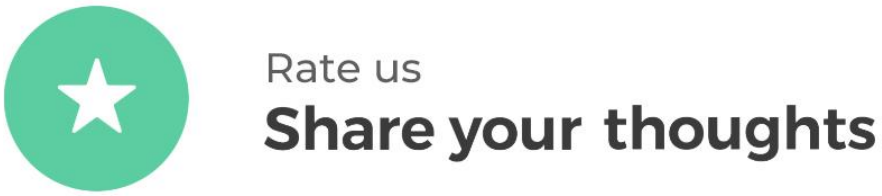
- Make sure that your application is attractively styled. Use any custom CSS and/or CSS libraries of your choice in this regard.

Compulsory Task 2

Use React to create an abridged online CV for yourself. For an example of what an online CV typically looks like, have a look at **this example**. Custom-styled CVs for developers are very common, and make a great impression on recruiters.

Feel free to spruce up your CV as you learn more React and web development in general. In a later task, you'll learn how to deploy your React app so the world can see it.

Remember to submit everything excluding the node_modules directory.



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.

REFERENCE

React.js. (2020). Getting Started – React. Retrieved 6 August 2020, from https://reactjs.org/docs/getting-started.html