React Advanced

Course syllabus

By the end of this reading, you will have learned about the scope of things you will cover in this course.

Prerequisites

To take this course, you should understand the basics of React, HTML, CSS, and JavaScript. Additionally, it always helps to have a can-do attitude!

Course content

This course covers advanced React development. You'll learn how to use more advanced React concepts and features, optimize and test your React applications, and become proficient in using React and JSX.

This course consists of four modules. They cover the following topics:

Module 1: Components

In this introductory module, you'll learn about React and your career opportunities. You'll also learn how to set up your coding environment so that you have as productive a learning experience as possible. The purpose of this module is to understand the what and the why behind learning React. You'll also learn about career opportunities and how to set up your coding environment for the most efficient and productive learning experience.

In React, everything revolves around components. You'll learn how to efficiently render list and form components, as well as how to lift up a shared state when several components need access to the data.

By the end of this module you will be able to:

- Outline React and various career opportunities.
- Render and transform lists with keys in React.
- Distinguish between controlled and uncontrolled React components.
- Create a controlled form component in React.
- Share component state by lifting state up to the closest common ancestor.
- Share global state using React Context.

Module 2: React Hooks and Custom Hooks

The second module of this course covers React hooks and custom hooks. You'll learn how to use all the common hooks in React, and how to put them to use within your application. You will also test your skills by building your own custom hooks.

By the end of this module you will be able to:

- Explain the uses and purpose of React hooks.
- Detail the concept and nature of state and state change.
- Use the State hook to declare, read and update the state of a component.
- Use the Effect hook to perform side-effects within a React component.
- Use hooks to fetch data in React.
- Create appropriate custom hooks in React.

Module 3: JSX and Testing

In this module, you'll learn about JSX and testing in React. You'll cover JSX in-depth and discover advanced patterns to encapsulate common behavior via Higher-order components and Render Props. You will then learn how to use Jest and the React Testing Library to write and perform tests on your applications.

By the end of this module, you will be able to:

- Define the types of children within JSX.
- Describe the process and purpose of creating render props.
- Describe the process and purpose of creating higher-order components.
- Use Jest and the React Testing Library to write and perform tests on your applications.

Module 4: Graded Assessment

In this module, you will be assessed on the key skills covered in the course and create a project to add to your portfolio.

You will be provided with code snippets, and your task will be to use these, plus any of your own code to complete your developers' portfolio.

This is a creative project, and the goal is to use as many React concepts as possible within this portfolio. You can use component composition, code reusability, hooks, manage state, interact with an external API, create forms, lists and so on.

By the end of this module, you will be able to:

- Synthesize the skills from this course to create a portfolio.
- Reflect on this course's content and on the learning path that lies ahead.

Setting up a React project in VS Code (Optional)

To complete the exercises in this course you have been provided with a dedicated lab environment set up specifically for you to apply the skills that you have learned. You can find out more about Working with Labs in this course in the following reading.

You can also use VS Code to practice these exercises on your local machine as an alternative option.

To follow along in this reading, you need to have Node.js and VS Code already installed on your computer. If you don't have this setup, please refer to the following resources in this course: Setting up VS Code and Installing Node and NPM.

In VS Code, you're ready to start a brand new React project.

You can do it using npm.

What is npm?

When Node.js is installed on a computer, **npm** comes bundled with it.

With **npm**, you can:

1. Author your own Node.js modules ("packages"), and publish them on the npm website so that other people can download and use them
2. Use other people's authored modules ("packages")

So, ultimately, npm is all about *code sharing* and *reuse*. You can **use other people's code** in your own projects, and you can also **publish your own Node.js modules** so that other people can use them.

An example npm module that can be useful for a new React developer is create-react-app. While this npm module comes with its own website, you can also find some info on the create-react-app project on GitHub.

Whenever you run the npm command to add other people's code, **that code, and all other Node modules that depend on it, get downloaded** to your machine.

However, although it's possible to do so, this is not really necessary, at least in the case of the `create-react-app` Node module.

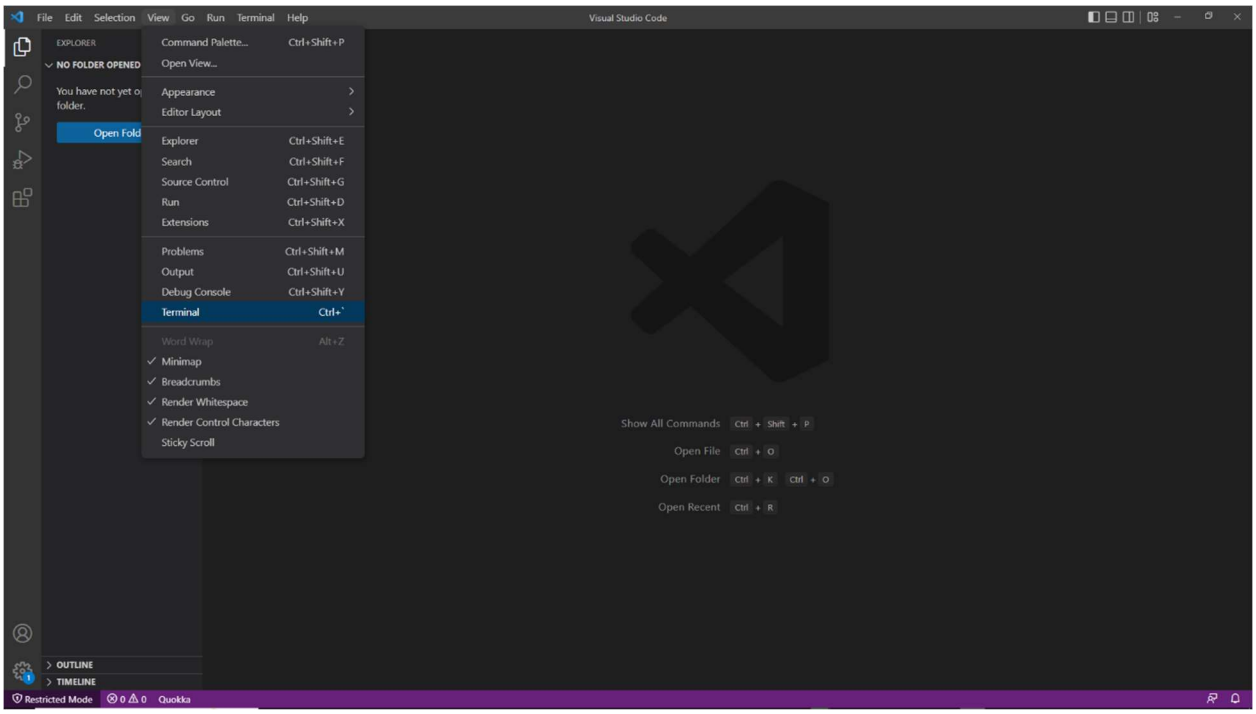
In other words, you can avoid installing the `create-react-app` package but still use it.

You can do that by running the following command: `npm init react-app example`, where "example" is the actual name of your app. You can use any name you'd like, but it's always good to have a name that is descriptive and short.

In the next section, you'll learn how to build a brand new app that you can name: **firstapp**.

Opening the built-in VS Code terminal and running *npm init react-app* command

In VS Code, click on *View*, *Terminal* to open the built-in terminal.



Now run the command to add a brand new React app to the machine:

```
npm init react-app firstapp
```

The installation and setup might take a few minutes.

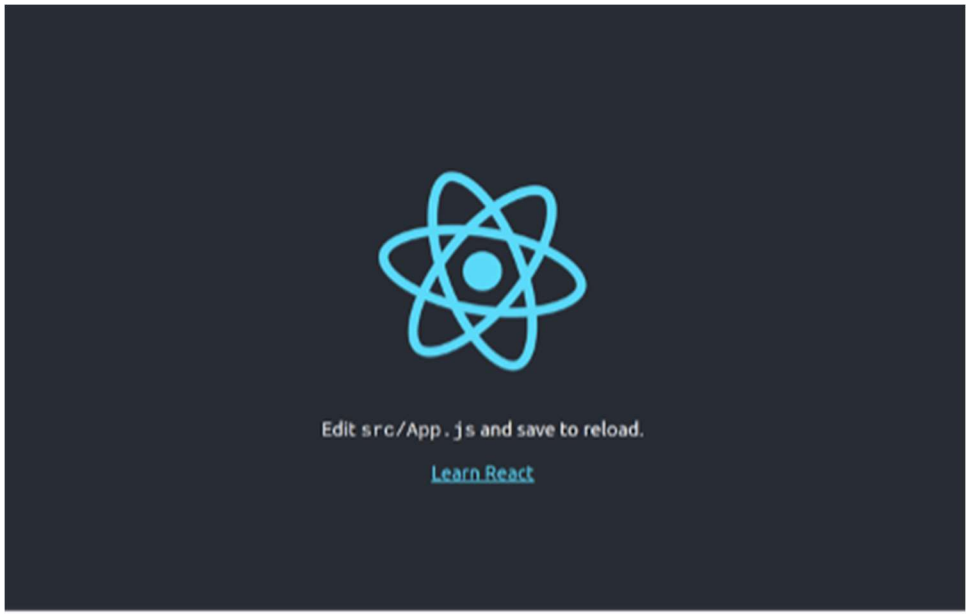
Here's the output of executing the above command:

```
Creating a new React app in /home/pc/Desktop/firstapp.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

added 1383 packages in 56s

190 packages are looking for funding
                      run `npm fund` for details

Initialized a git repository.
```

```
Installing template dependencies using npm...
npm WARN deprecated source-map-resolve@0.6.0:
See https://github.com/lydell/source-map-resolve#deprecated

added 39 packages in 6s

190 packages are looking for funding
                        run `npm fund` for details
Removing template package using npm...

removed 1 package, and audited 1422 packages in 3s

190 packages are looking for funding
                        run `npm fund` for details

6 high severity vulnerabilities

To address all issues (including breaking changes), run:
npm audit fix --force

Run `npm audit` for details.

Created git commit.

Success! Created firstapp at /home/pc/Desktop/firstapp

Inside that directory, you can run several commands:
```

If you follow the suggestions from the above output, you'll run: **`cd firstapp`**, and then **`npm start`**.

Following the instructions, opening a browser with the address bar pointing to http://localhost:3000, will show the following page in your browser:



This means that you've successfully:

- Set up your local development environment
- Run the create-react-app npm package (without installing it!)
- Built a starter React app on your local machine
- Served that starter React app in your browser

After you've built your starting setup, in Module 2 you'll start working with the basic building blocks of React: components.

RENDERING LISTS IN REACT

Exercise: Create a basic List component

Instructions

Task

The Little Lemon restaurant has decided to remove all desserts with high calories from their menu.

In this lab, you are going to implement a new list component, **`DessertsList`**, that will display a list of desserts with less than 500 calories, all sorted by calories, from low to high.

The data you have to work with has been provided to you inside the App.js file, as an array of objects. Each object represents a dessert and has the following properties: **`name`**, **`calories`** and **`createdAt`**.

The **`App`** component passes that information to the **`DessertsList`** component as a prop named **`data`**.

Each item from the list should display the name of the dessert and the number of calories, both separated by a dash character, i.e. **`Chocolate Mousse - 250 cal`**.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

If you run **`npm start`** and view the app in the browser, you'll notice that the starting React app works as is. The app outputs the below interface with a simple header. You'll build from that starting point.

Steps

Step 1
Open the **`DessertsList.js`** file. You only need to implement this component to complete this exercise.

You'll see an empty component that at the moment returns `null`, resulting in rendering nothing at all.

The `DessertsList` component receives a prop called `data`, which is an array containing the list of desserts. You can check the exact shape of the data at the top of the `App.js` file, under a variable called `desserts`.

Step 2

Inside the `DessertsList` component, remove the `null` and instead return a `ul` element that contains a list of `li` elements, where each `li` text is a dessert with the following format: `${dessertName} - ${dessertCalories} cal`.

The list should be sorted by calories in an ascending manner and any desserts with more than 500 calories should be excluded. For that you have to use a combination of `map`, `filter` and `sort` array operators.

Step 3

Save all the changes and run the app. Preview the updates in the browser, and confirm that the page shows an `ul` element with just three `li` elements as below:

- Ice Cream - 200 cal
- Tiramisu - 300 cal
- Chocolate Cake - 400 cal

Tip

If you're having trouble with this lab, please review the `filter` and `sort` methods from arrays in JavaScript.

Solution: Create a basic List component

There are three types of operations you need to apply to the list of desserts: filtering, sorting and mapping.

Although the order of the operations is not that important, it's recommended to leave the final projection (mapping) to the end, since that final projection may skip some of the data needed for the filtering and sorting criteria.

Important note

If your browser doesn't automatically reload with the updates, please remember to manually refresh your browser to view the changes after modifying **DessertsList.js**.

Filtering

The first requirement is to display desserts that have less than 500 calories. That means Cheesecake, which has 600 cal, should be omitted. When you need to eliminate elements from your lists based on a certain condition or set of conditions, you need to use the `filter()` method.

The `filter` method creates a copy of the array, filtered down to just the elements from the original array that pass the test. In order words, it will return a new list with just the elements that fulfil the condition.

Each dessert from the list has a property called `calories`, which is an integer representing the number of calories. Therefore, the condition to be implemented should be as follows:

```
const lowCaloriesDesserts = props.data
 .filter((dessert) => {
   return dessert.calories < 500;
 })
```

`lowCaloriesDessert` variable will then hold a list of three desserts, without Cheesecake.

Sorting

The second requirement you have to implement is sorting the list by calories, from low to high or in ascending order. For that, arrays in JavaScript offer the `sort()` method, which sorts the elements of an array based on a comparison function provided. The `return` value from that comparison function determines how the sorting is performed:

compareFn(a, b) return value	sort order
`> 0`	sort **a** after **b**
`< 0`	sort **a** before **b**
`=== 0`	keep original order of **a** and **b**

You can chain one operation after another. Recall that `filter` returns the new array with the filtered down elements, so sort can be chained right after that, as below:

```
const lowCaloriesDesserts = props.data
 .filter((dessert) => {
   return dessert.calories < 500;
 })
 .sort((a, b) => {
   return a.calories - b.calories;
 })
```

The compare function makes sure the sorting occurs in ascending order, according to the table above.

Mapping

Finally, to apply the desired projection and display the information as requested, you can chain the `map` operator at the end and return a `<li>` item with the dessert name and its calories, both separated by a dash character, and the word “cal” at the end.

The final code should look like below:

```
const lowCaloriesDesserts = props.data
```

```
 .filter((dessert) => {
   return dessert.calories < 500;
 })
 .sort((a, b) => {
   return a.calories - b.calories;
 })
 .map((dessert) => {
   return (
     <li>
       {dessert.name} - {dessert.calories} cal
     </li>
   );
 });
```

And the full implementation of the DessertsList component:

```
const DessertsList = (props) => {
 const lowCaloriesDesserts = props.data
   .filter((dessert) => {
     return dessert.calories < 500;
   })
   .sort((a, b) => {
     return a.calories - b.calories;
   })
   .map((dessert) => {
     return (
       <li>
         {dessert.name} - {dessert.calories} cal
       </li>
     );
   });
 return <ul>{lowCaloriesDesserts}</ul>;

}
export default DessertsList;
```

Final result

This is what should be displayed in your browser:

List of low calorie desserts:

- Ice Cream - 200 cal
- Tiramisu - 300 cal
- Chocolate Cake - 400 cal

Additional resources

Here is a list of additional resources for Rendering Lists in React:

- Map() allows you to create new arrays populated with the results of calling a transformation function on every element.
- Rendering lists on official React docs website dives deeper into how to display multiple similar components from a collection of data, providing examples of both filtering and transformations.
- React keys on official docs offers a comprehensive set of memotecnic rules to reinforce how to use keys properly.

FORMS IN REACT

Controlled components vs. Uncontrolled components

This reading will teach you how to work with uncontrolled inputs in React and the advantages of controlled inputs via state design. You will also learn when to choose controlled or uncontrolled inputs and the features each option supports.

Introduction

In most cases, React recommends using controlled components to implement forms. While this approach aligns with the React declarative model, uncontrolled form fields are still a valid option and have their merit. Let's break them down to see the differences between the two approaches and when you should use each method.

Uncontrolled Inputs

Uncontrolled inputs are like standard HTML form inputs:

```
const Form = () => {
 return (
   <div>
     <input type="text" />
   </div>
 );
};
```

They remember exactly what you typed, being the DOM itself that maintains that internal state. How can you then get their value? The answer is by using a React ref.

In the code below, you can see how a ref is used to access the value of the input whenever the form is submitted.

```
const Form = () => {
 const inputRef = useRef(null);

 const handleSubmit = () => {
   const inputValue = inputRef.current.value;
   // Do something with the value
 }
 return (
   <form onSubmit={handleSubmit}>
     <input ref={inputRef} type="text" />
   </form>
```

```
 );
};
```

In other words, you must **pull** the value from the field when needed.

Uncontrolled components are the simplest way to implement form inputs. There are certainly valued cases for them, especially when your form is straightforward. Unfortunately, they are not as powerful as their counterpart, so let's look at controlled inputs next.

Controlled Inputs

Controlled inputs accept their current value as a prop and a callback to change that value. That implies that the value of the input has to live in the React state somewhere. Typically, the component that renders the input (like a form component) saves that in its state:

```
const Form = () => {
 const [value, setValue] = useState("");

 const handleChange = (e) => {
   setValue(e.target.value)
 }

 return (
   <form>
     <input
       value={value}
       onChange={handleChange}
       type="text"
     />
   </form>
 );
};
```

Every time you type a new character, the `handleChange` function is executed. It receives the new value of the input, and then it sets it in the state. In the code example above, the flow would be as follows:

- The input starts out with an empty string: `""`
- You type `"a"` and `handleChange` gets an `"a"` attached in the event object, as `e.target.value`, and subsequently calls `setValue` with it. The input is then updated to have the value of `"a"`.
- You type `"b"` and `handleChange` gets called with `e.target.value` being `"ab"`.and sets that to the state. That gets set into the state. The input is then re-rendered once more, now with `value = "ab"` .

This flow **pushes** the value changes to the form component instead of pulling like the ref example from the uncontrolled version. Therefore, the Form component always has the input's current value without needing to ask for it explicitly.

As a result, your data (React state) and UI (input tags) are always in sync. Another implication is that forms can respond to input changes immediately, for example, by:

- Instant validation per field
- Disabling the submit button unless all fields have valid data
- Enforcing a specific input format, like phone or credit card numbers

Sometimes you will find yourself not needing any of that. In that case uncontrolled could be a more straightforward choice.

The file input type

There are some specific form inputs that are always uncontrolled, like the file input tag.

In React, an `<input type="file" />` is always an uncontrolled component because its value is read-only and can't be set programmatically.

The following example illustrates how to create a ref to the DOM node to access any files selected in the form submit handler:

```
const Form = () => {
 const fileInput = useRef(null);

 const handleSubmit = (e) => {
   e.preventDefault();
   const files = fileInput.current.files;
   // Do something with the files here
 }

 return (
   <form onSubmit={handleSubmit}>
     <input
       ref={fileInput}
       type="file"
     />
   </form>
 );
};
```

Conclusion

Uncontrolled components with refs are fine if your form is incredibly simple regarding UI feedback. However, controlled input fields are the way to go if you need more features in your forms.

Evaluate your specific situation and pick the option that works best for you.

The below table summarizes the features that each one supports:

Feature	Uncontrolled	Controlled
One-time value retrieval (e.g. on submit)	**Yes**	**Yes**
Validating on submit	**Yes**	**Yes**
Instant field validation	No	**Yes**
Conditionally disabling a submit button	No	**Yes**
Enforcing a specific input format	No	**Yes**
Several inputs for one piece of data	No	**Yes**
Dynamic inputs	No	**Yes**

And that's it about controlled vs. uncontrolled components. You have learned in detail about each option, when to pick one or another, and finally, a comparison of the features supported.

Exercise: Create a registration form

Task

You've learned how to create controlled components and forms in React. Now it's time to put that knowledge to use and create a registration form for Little Lemon Restaurant, where users are able to sign up.

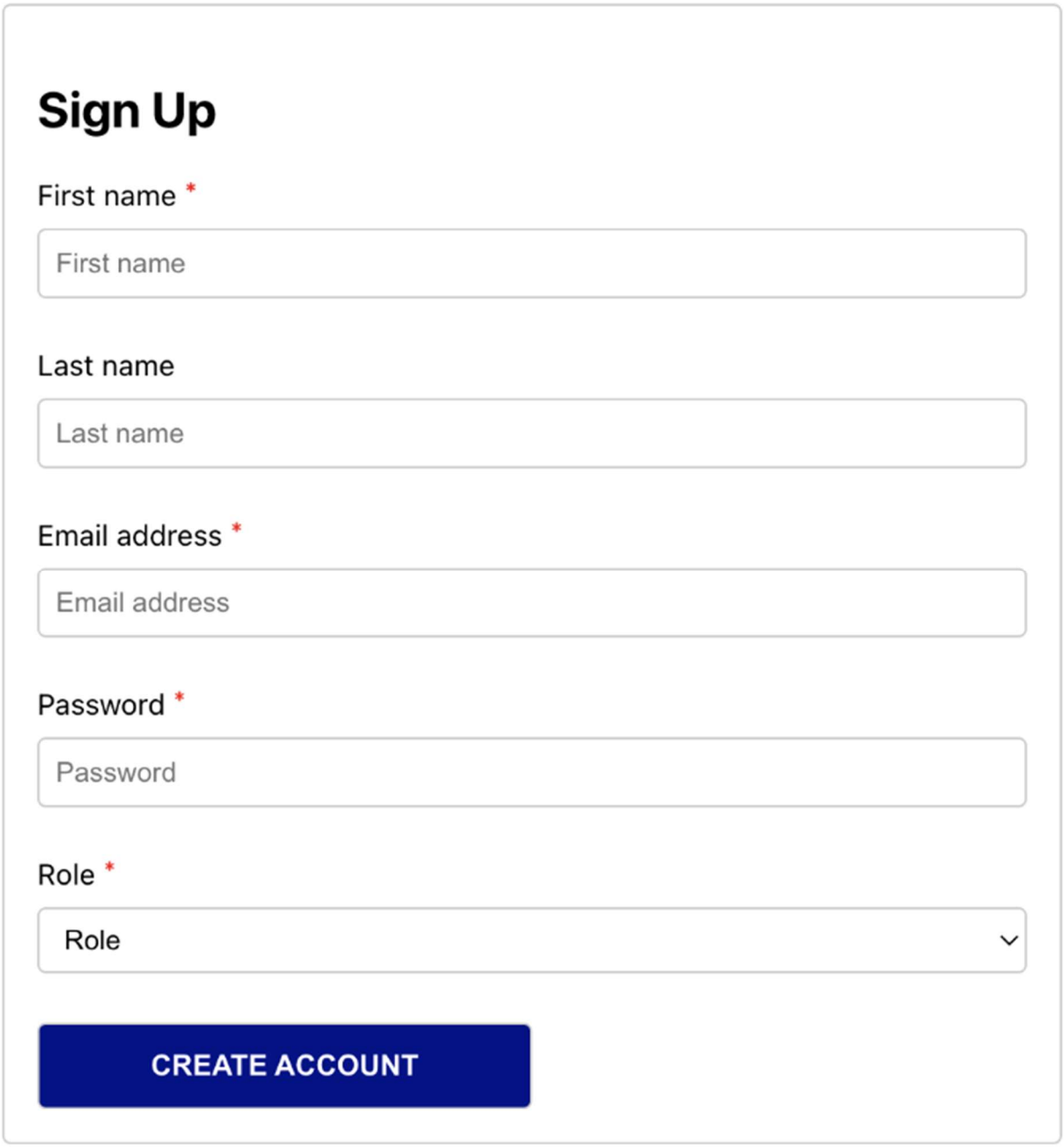
The form layout and styling is already predefined for you. You have to add the missing pieces of code to make the form work as per the requirements. The form is provided in an uncontrolled fashion and contains the following inputs:

- First Name
- Last Name
- Email
- Password
- Role

All the local state needed to complete this task has been already defined for you.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

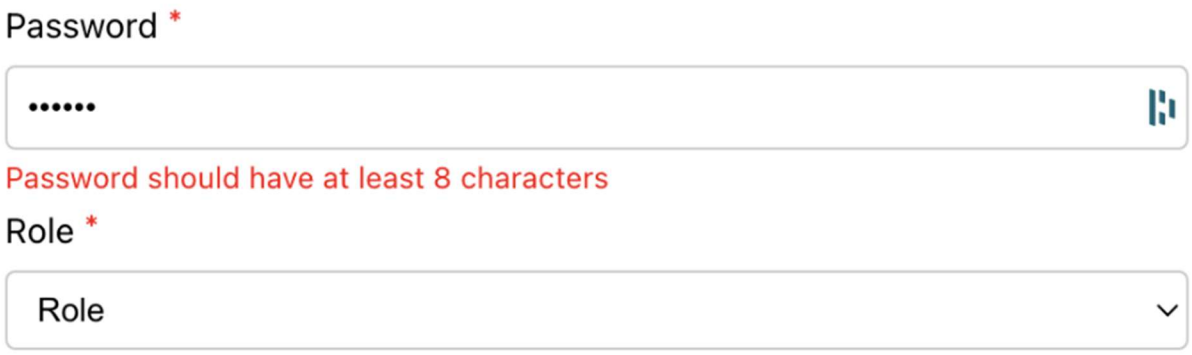
If you run `npm start` and view the app in the browser you'll notice that the starting React app works as is. The app outputs a sign up form with five different fields and a submit button.



Steps

Step 1
Open the **`App.js`** file. Convert all the elements from the form to controlled components by adding the value and **`onChange`** attributes to each input. Make sure the password input is obscured.

Step 2
Show an error message if the password is less than 8 characters long and the user has interacted with the input at least once. The error message should be displayed below the password input as follows.



For that, a component called **`PasswordErrorMessage`** has been provided to you. Your goal is to implement the logic for when to show the error message. The password state is a special one that has an additional property called **`isTouched`**. This property is used to determine if the user has interacted with the input or not.

Step 3
Prevent the default behaviour of the form when the user clicks the submit button.

Step 4
Implement the body of **`getIsFormValid`** function to return **`true`** if the form is valid and **`false`** otherwise. This determines the submit button state. The rules for the form to be valid are as follows:

- The first name cannot be empty.
- The email must be a valid email address and can't be empty. A function called **`validateEmail`** has already been provided for you to check if the email is valid. It returns **`true`** if the email is valid, otherwise **`false`** is returned.
- The password must be at least 8 characters long.
- The role must be either **`individual`** or **`business`**.

Step 5
Implement the body of **`clearForm`** function to clear the form state after a successful submission.

Tip
React offers two focus events for form elements: **`onBlur`** and **`onFocus`**. You can check more information about them here.

Solution: Create a registration form

Here is the completed solution code for the App.js file:

```
import './App.css';
import {useState} from "react";
import {validateEmail} from "../src/utils";

const PasswordErrorMessage = () => {
 return (
   <p className="FieldError">Password should have at least 8 characters</p>
 );
};

function App() {
 const [firstName, setFirstName] = useState("");
 const [lastName, setLastName] = useState("");
 const [email, setEmail] = useState("");
 const [password, setPassword] = useState({
   value: "",
   isTouched: false,
 });
 const [role, setRole] = useState("role");

 const getIsFormValid = () => {
   return (
     firstName &&
     validateEmail(email) &&
     password.value.length >= 8 &&
     role !== "role"
   );
 };

 const clearForm = () => {
   setFirstName("");
   setLastName("");
   setEmail("");
   setPassword({
     value: "",
     isTouched: false,
   });
   setRole("role");
 };
```

```
const handleSubmit = (e) => {
  e.preventDefault();
  alert("Account created!");
  clearForm();
};

return (
  <div className="App">
    <form onSubmit={handleSubmit}>
      <fieldset>
        <h2>Sign Up</h2>
        <div className="Field">
          <label>
            First name <sup>*</sup>
          </label>
          <input
            value={firstName}
            onChange={(e) => {
              setFirstName(e.target.value);
            }}
            placeholder="First name"
          />
        </div>
        <div className="Field">
          <label>Last name</label>
          <input
            value={lastName}
            onChange={(e) => {
              setLastName(e.target.value);
            }}
            placeholder="Last name"
          />
        </div>
        <div className="Field">
          <label>
            Email address <sup>*</sup>
          </label>
          <input
            value={email}
            onChange={(e) => {
              setEmail(e.target.value);
            }}
            placeholder="Email address"
          />
```

```
          </div>
          <div className="Field">
            <label>
              Password <sup>*</sup>
            </label>
            <input
              value={password.value}
              type="password"
              onChange={(e) => {
                setPassword({ ...password, value: e.target.value });
              }}
              onBlur={() => {
                setPassword({ ...password, isTouched: true });
              }}
              placeholder="Password"
            />
            {password.isTouched && password.value.length < 8 ? (
              <PasswordErrorMessage />
            ) : null}
          </div>
          <div className="Field">
            <label>
              Role <sup>*</sup>
            </label>
            <select value={role} onChange={(e) => setRole(e.target.value)}>
              <option value="role">Role</option>
              <option value="individual">Individual</option>
              <option value="business">Business</option>
            </select>
          </div>
          <button type="submit" disabled={!getIsFormValid()}>
            Create account
          </button>
        </fieldset>
      </form>
    </div>
  );
}

export default App;
```

Step 1

The first step involves converting all form elements into controlled components. Since the pieces of local state have been already defined at the top of the component, you just have to assign each state piece to the **`value`** prop from each input element. To be able to account for state updates, each input should also define the **`onChange`** prop and call the state setter with the value property from the event target as parameter.

The password input is a special case that has an object as state instead of a string. As a result, the state setter should spread the previous values so they don't get overridden. Finally, to make sure the password characters are obscured, you need to use the type "password" for the input.

```
<div className="Field">
 <label>
   First name <sup>*</sup>
 </label>
 <input
   value={firstName}
   onChange={(e) => {
     setFirstName(e.target.value);
   }}
   placeholder="First name"
 />
</div>
<div className="Field">
 <label>Last name</label>
 <input
   value={lastName}
   onChange={(e) => {
     setLastName(e.target.value);
   }}
   placeholder="Last name"
 />
</div>
<div className="Field">
 <label>
   Email address <sup>*</sup>
 </label>
 <input
   value={email}
   onChange={(e) => {
     setEmail(e.target.value);
   }}
   placeholder="Email address"
 />
</div>
<div className="Field">
```

```
 <label>
   Password <sup>*</sup>
 </label>
 <input
   value={password.value}
```

Step 2

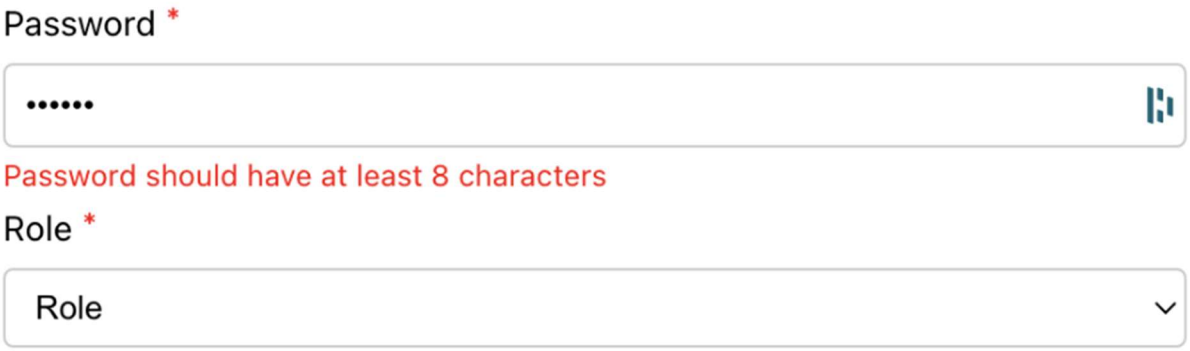
The **`isTouched`** property on the password state was defined to determine when the input was touched at least once. In order to listen for interactions, form inputs have two additional events you can subscribe to: **`onFocus`** and **`onBlur`**.

In this scenario, you need to use the **`onBlur`** event, which is called whenever the input loses focus, so that guarantees the user has interacted with the password input at least once. In that event, you should set the **`isTouched`** property to **`true`** with the password state setter.

Then, the condition to display the error message relies on that value being **`true`** and a check on the password length to see if it's less than 8 characters long. If the condition is **`true`**, the component **`PasswordErrorMessage`** should be rendered. The final code should be as follows:

```
<div className="Field">
 <label>
   Password <sup>*</sup>
 </label>
 <input
   value={password.value}
   type="password"
   onChange={(e) => {
     setPassword({ ...password, value: e.target.value });
   }}
   onBlur={() => {
     setPassword({ ...password, isTouched: true });
   }}
   placeholder="Password"
 />
 {password.isTouched && password.value.length < 8 ? (
   <PasswordErrorMessage />
 ) : null}
</div>
```

If implemented correctly, the form should display an error message below the password field:



Step 3

To prevent the default behavior of the form when clicking on the submit button, you have to call **`preventDefault`** on the event object, right in your submit handler function.

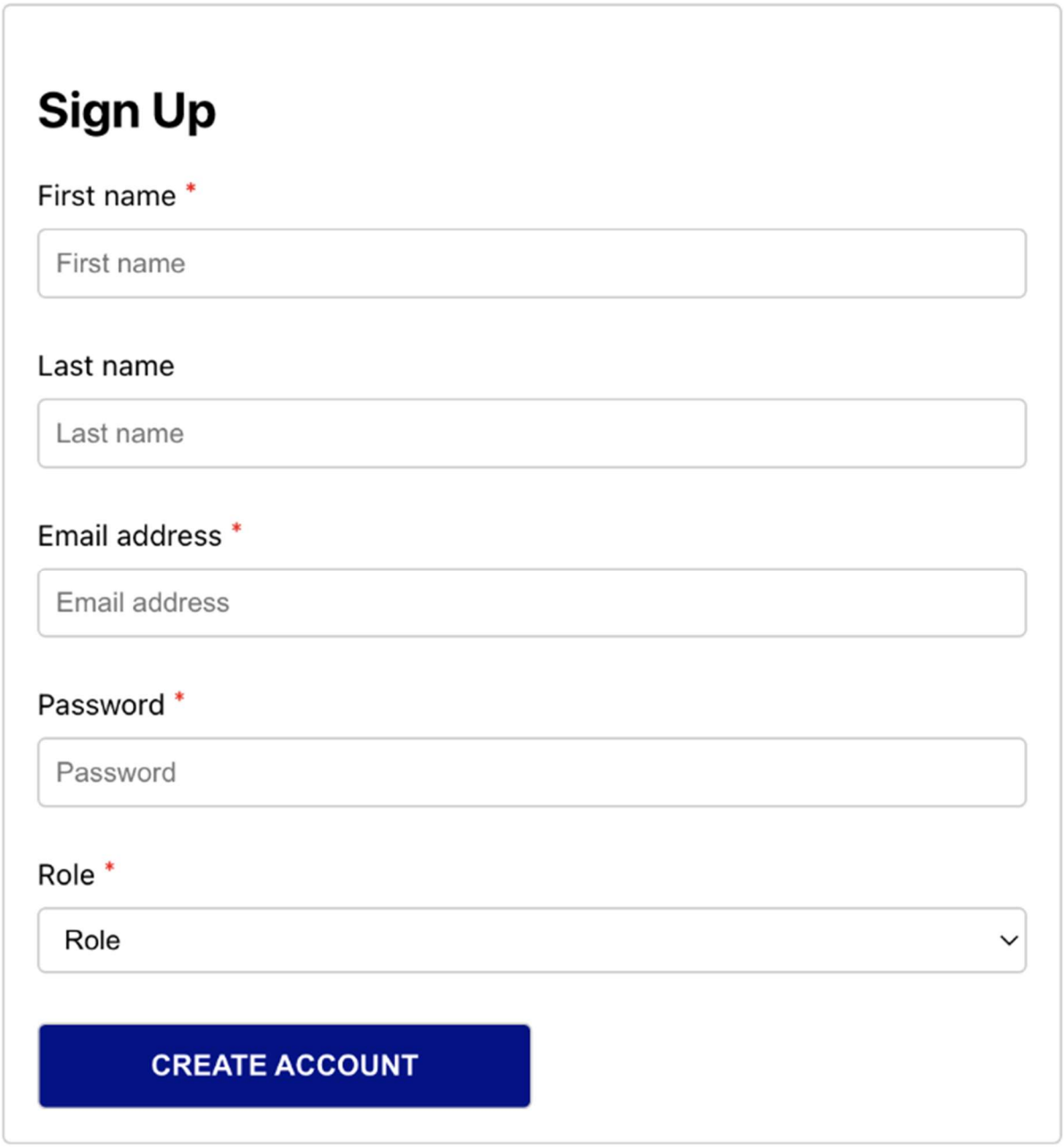
```
const handleSubmit = (e) => {
 e.preventDefault();
 alert("Account created!");
 clearForm();
};
```

Step 4

To fulfil the validation rules of the form, the body of the **`getIsFormValid`** function should be implemented as below:

```
const getIsFormValid = () => {
 return (
   firstName &&
   validateEmail(email) &&
   password.value.length >= 8 &&
   role !== "role"
 );
};
```

Below is an example of a valid form:



Step 5

Finally, to clear the form state after a successful submission, you should set each piece of state to its initial value:

```
const clearForm = () => {
 setFirstName("");
 setLastName("");
```

```
 setEmail("");
 setPassword({
   value: "",
   isTouched: false,
 });
 setRole("role");
};
```

Now, when you submit the form, an alert is displayed and right after you dismiss it, the form is reset to its initial values.

Additional resources

Here is a list of additional resources for Module 1, Lesson 3 (Forms in React):

- Forms from the official React docs illustrate some examples of how React deals with certain form fields compared to traditional HTML tags, like the text area, select and file input tags. It also showcases how to handle multiple inputs by leveraging **`event.target.name`** and the implications of using null as a value in a controlled input.
- Formik is the most popular open source form library for React. It saves you lots of time when building forms and offers a declarative, intuitive and adoptable paradigm.
- Yup is an open-source library that integrates perfectly with Formik. It allows you to set all your form validation rules declaratively.
- React-hook-form is another popular library to easily manage your form state and validation rules.

REACT CONTEXT

Exercise: Create a light-dark theme switcher

Instructions

Task

You've learned about React Context and how it allows you to define global state without passing individual props down through each component. One of the most common use cases for Context is to define a theme for your application. In this exercise, you'll create a light-dark theme switcher.

The starter code includes all the necessary UI elements, as well as a **`Switch`** component to toggle the theme. Your goal is to implement the missing functionality inside **`ThemeContext.js`**. **`ThemeContext`** already exports a **`ThemeProvider`** component and a **`useTheme`** hook. At the moment, they don't do anything and return dummy values.

You'll need to implement both the **`ThemeProvider`** component and the **`useTheme`** hook inside the **`ThemeContext.js`** file to complete this exercise.

`ThemeProvider` should render a context provider component and inject as the context value an object with two properties: a **`theme`** property that is a string, that can be either **`light`** or **`dark`**, and a function named **`toggleTheme`** that enables you to toggle the theme. The **`useTheme`** hook should return that context object.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

If you run **`npm start`** and view the app in the browser, you'll notice that the starting React app works as is. The app outputs a simple view with a header, page and a switch widget in the top right corner to change the theme.

Steps

Step 1
Open the **`ThemeContext.js`** file.

Create a **`ThemeContext`** object using **`React.createContext()`**.

Implement the **`ThemeProvider`** component. It should accept a **`children`** prop and return a **`ThemeContext.Provider`** component. The **`ThemeContext.Provider`** receives an object as its **`value`** prop, with a **`theme`** string and a **`toggleTheme`** function.

`toggleTheme` should toggle the theme between **`light`** and **`dark`**.

Step 2
Implement the **`useTheme`** hook. It should return the **`theme`** and **`toggleTheme`** values from the **`ThemeContext`**.

Step 3
Open the **`Switch/index.js`** file. Add an **`onChange`** prop to the input element and pass a callback function, as the event handler, to change the theme. You don't need to use the event argument in this case.

Step 4
Verify that the app works as expected. You should be able to toggle the theme between light and dark. Notice how the background color of the page changes, as well as the color of the text.

Solution: Create a light-dark theme switcher

Here is the completed solution code for the **ThemeContext.js** file:

```
import { createContext, useContext, useState } from "react";

const ThemeContext = createContext(undefined);

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider
      value={{
        theme,
        toggleTheme: () => setTheme(theme === "light" ? "dark" : "light"),
      }}
    >
      {children}
    </ThemeContext.Provider>
  );
};

export const useTheme = () => useContext(ThemeContext);
```

Here is the solution code for the **Switch/index.js** file:

```
import "./Styles.css";
import { useTheme } from "../ThemeContext";

const Switch = () => {
 const { theme, toggleTheme } = useTheme();
 return (
   <label className="switch">
     <input
       type="checkbox"
       checked={theme === "light"}
       onChange={toggleTheme}
     />
     <span className="slider round" />
   </label>
 );
};

export default Switch;
```

Steps

Step 1

To create the **`ThemeProvider`**, the first step is to create a new context object, **`ThemeContext`**, using **`createContext`**, a function that can be imported from React. The default value argument is only used when a component does not have a matching Provider above it in the tree. This default value can be helpful for testing components in isolation without wrapping them. For the purpose of this exercise, it's not relevant, so **`undefined`** can be used.

Then, inside the **`ThemeProvider`** component, you need to define a new piece of local state for the theme, which can be a string whose value is either **`"light"`** or **`"dark"`**. It can be initialized to **`"light"`**, which is usually the default theme for applications.

In the **`return`** statement, the **`ThemeContext.Provider`** component should be rendered and wrap the children.

Finally, recall that the value prop for **`ThemeContext.Provider`** is what gets injected down the tree as context. Since the application needs both the theme value and a way to toggle it, two values are injected: **`theme`** and **`toggleTheme`**.

`theme` is just the light-dark theme string value, whereas **`toggleTheme`** is a function that receives no parameters and just toggles the theme from light to dark and vice versa.

That completes the implementation of the **`ThemeProvider`** component, as per the code below:

```
import { createContext, useContext, useState } from "react";

const ThemeContext = createContext(undefined);

export const ThemeProvider = ({ children }) => {
 const [theme, setTheme] = useState("light");

 return (
   <ThemeContext.Provider
     value={{
       theme,
       toggleTheme: () => setTheme(theme === "light" ? "dark" : "light"),
     }}
   >
     {children}
   </ThemeContext.Provider>
```

```
 );
};
```

Step 2

The implementation for **`useTheme`** is quite simple. You just need to import the **`useContext`** hook from React and pass as an argument the **`ThemeContext`** object defined before. That allows your components to access both **`theme`** and **`toggleTheme`** values, which are the ones the **`useTheme`** custom hook returns.

```
1
export const useTheme = () => useContext(ThemeContext);
```

Step 3

The **`Switch`** component can then be connected to the **`toggleTheme`** function returned from **`useTheme`** as per the code below:

```
const Switch = () => {
 const { theme, toggleTheme } = useTheme();
 return (
   <label className="switch">
     <input
       type="checkbox"
       checked={theme === "light"}
       onChange={toggleTheme}
     />
     <span className="slider round" />
   </label>
 );
};
```

Step 4

And, finally, you should be able to use the switch widget on the top right corner to change the theme of the application:

Additional resources

Here is a list of additional resources for React Context.

- React.memo from the official React docs, an API that can be used in conjunction with Context Providers to prevent unnecessary re-renders in top-level components in the tree.
- useMemo from the official React docs, a hook to guarantee referential equality on objects across rendering passes.

WEEK 2 - GETTING START WITH HOOKS

Working with complex data in useState

In this reading, you will learn how to use objects as state variables when using **`useState`**. You will also discover the proper way to only update specific properties, such as state objects and why this is done. This will be demonstrated by exploring what happens when changing the string data type to an object.

An example of holding state in an object and updating it based on user-generated events

When you need to hold state in an object and update it, initially, you might try something like this:

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


export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);


  function updateGreeting() {
    setGreeting({ greet: "Hello, World-Wide Web" });
  }


  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

While this works, it's not the recommended way of working with state objects in React, this is because the state object usually has more than a single property, and it is costly to update the entire object just for the sake of updating only a small part of it.

The correct way to update the state object in React when using useState

The suggested approach for updating the state object in React when using **`useState`** is to copy the state object and then update the copy.

This usually involves using the spread operator (`...`).

Keeping this in mind, here's the updated code:

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


export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);


  function updateGreeting() {
    const newGreeting = {...greeting};
    newGreeting.greet = "Hello, World-Wide Web";
    setGreeting(newGreeting);
  }


  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

Incorrect ways of trying to update the state object

To prove that a copy of the old state object is needed to update state, let's explore what happens when you try to update the old state object directly:

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


export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);
```

```
  function updateGreeting() {
    greeting = {greet: "Hello, World-Wide Web};
    setGreeting(greeting);
  }

  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

The above code does not work because it has a **`TypeError`** hiding inside of it.

Specifically, the **`TypeError`** is: "Assignment to constant variable".

In other words, you cannot reassign a variable declared using const, such as in the case of the **`useState`** hook's array destructuring:

1

```
const [greeting, setGreeting] = useState({ greet: "Hello, World" });
```

Another approach you might attempt to use to work around the suggested way of updating state when working with a state object might be the following:

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

export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);

  function updateGreeting() {
    greeting.greet = "Hello, World-Wide Web;
```

```
    setGreeting(greeting);
  }


  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

The above code is problematic because it doesn't throw any errors; however, it also doesn't update the heading, so it is not working correctly. This means that, regardless of how many times you click the "Update greeting" button, it will still be "Hello, World".

To reiterate, the proper way of working with state when it's saved as an object is to:

1. Copy the old state object using the spread (`...`) operator and save it into a new variable and
2. Pass the new variable to the state-updating function

Updating the state object using arrow functions

Now, let's use a more complex object to update state.

The state object now has two properties: greet and location.

The intention of this update is to demonstrate what to do when only a specific property of the state object is changing, while keeping the remaining properties unchanged:

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


export default function App() {
  const [greeting, setGreeting] = useState(
    {
        greet: "Hello",
        place: "World"
    }
  );
```

```
  console.log(greeting, setGreeting);


  function updateGreeting() {
    setGreeting(prevState => {
        return {...prevState, place: "World-Wide Web"}
    });
  }


  return (
    <div>
      <h1>{greeting.greet}, {greeting.place}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

The reason this works is because it uses the previous state, which is named **`prevState`**, and this is the previous value of the greeting variable. In other words, it makes a copy of the **`prevState`** object, and updates only the place property on the copied object. It then returns a brand-new object:

```
return {...prevState, place: "World-Wide Web"}
```

Everything is wrapped in curly braces so that this new object is built correctly, and it is returned from the call to **`setGreeting`**.

Conclusion

You have learned what happens when changing the string data type to an object, with examples of holding state in an object and updating it based on user-generated events. You also learned about correct and incorrect ways to update the state object in React when using **`useState`**, and about updating the state object using arrow functions.

Exercise: Managing state within a component

Instructions

Note: For an introduction on how to work with labs in this course, please refer to the reading titled Working with Labs in this course.

Task

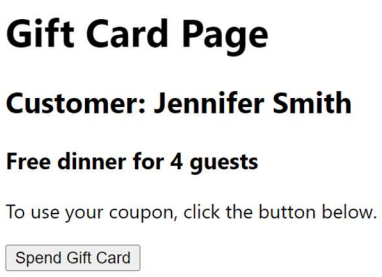
You've revised the `useState` hook. You've also learned about working with primitive (string) data and with complex data (state stored in objects).

In this code lab, you'll practice updating the state stored in an object, based on the user interacting with the app.

This code lab's app shows a Gift Card page of the Little Lemon Restaurant web app, where a visitor initially has a Gift Card that they can use to have a dinner for four.

Note: If you run `npm start` and view the app in the browser you'll notice that the starting React app works as is.

The starter code shows the following information on the screen:



In other words, the text that shows on the screen is as follows:

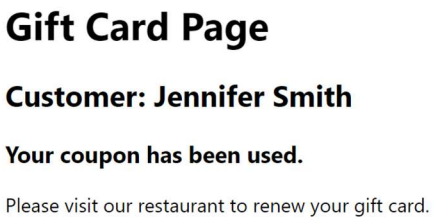
Gift Card Page Customer: Jennifer Smith Free dinner for 4 guests To use your coupon, click the button below. Spend Gift Card

The "Spend Gift Card" button is set up to execute a function when clicked. However, that event-handling function is empty.

That means that when serving the app with the starter code, if you click the "Spend Gift Card" button, there will be no change on the screen.

Your task is to complete the event-handling function for the "Spend Gift Card" button clicks, as detailed in the steps below.

When the code lab is successfully completed, after the "Spend Gift Card" button is clicked, the UI should update to show the following information on the screen:



In other words, the text that shows on the screen is as follows:

Gift Card Page Customer: Jennifer Smith Your coupon has been used. Please visit our restaurant to renew your gift card.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

Steps

Step 1

Open the `App.js` file.

Locate the `spendGiftCard()` function.

Inside the `spendGiftCard()` function, invoke the `setGiftCard()` state-updating function. For now, just invoke it, without passing it any parameters or doing anything else with it.

Step 2

Inside the `setGiftCard()` function invocation's parentheses, pass in an arrow function.

This arrow function has a single parameter, named `prevState`. After the arrow, add a block of code (starting with an opening curly brace, and ending with a closing curly brace two lines below).

Step 3

In Step 2, you've added the previous state object as the `prevState` argument of the arrow function you passed to the `setGiftCard()` function.

Now you need to return a new object based on this previous state object.

For now, in Step 3, you need to just return a copy of the `prevState` object.

That means that you need to use the **`return`** keyword and a copy of the **`prevState`** object, using the rest operator - that is, the **`...`** operator.

Step 4

In Step 3, you returned a copy of the **`prevState`** object using the rest operator.

Now you need to combine this copy of the **`prevState`** object with those properties that you want updated.

Put differently, you need to update some of the key-value pairs that already exist on the state object that were initially passed to the **`useState()`** function call.

For now, in Step 4, update the **`text`** property of the state object, as follows:

```
text: "Your coupon has been used."
```

Step 5

In Step 4, you've updated the **`text`** property on the state object.

In this step, you need to update the remaining properties on the state object.

You need to update the **`valid`** key's value to **`false`**.

You need to update the **`instructions`** key's value to **`Please visit our restaurant to renew your gift card.`**.

Step 6

Save the changes and run the app.

Verify that the completed app behaves as follows:

1. Initially, the Spend Gift Card button is showing.
2. Once you click the Spend Gift Card button, the **`text`** property value's update will now show the sentence that reads "Your coupon has been used".
3. Additionally, the **`instructions`** key's value's update will now show the text that reads "Please visit our restaurant to renew your gift card."
4. Finally, since the **`valid`** key's value was updated to **`false`**, the button is no longer showing.

Conclusion

In this exercise, you've practiced managing state within a component.

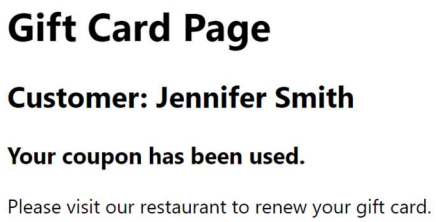
Solution: Managing state within a component

Here is the completed solution code for the **`App.js`** file:

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


export default function App() {
  const [giftCard, setGiftCard] = useState(
    {
        firstName: "Jennifer",
        lastName: "Smith",
        text: "Free dinner for 4 guests",
        valid: true,
        instructions: "To use your coupon, click the button below.",
    }
  );

  function spendGiftCard() {
    setGiftCard(prevState => {
        return {
          ...prevState,
          text: "Your coupon has been used.",
          valid: false,
          instructions: "Please visit our restaurant to renew your gift card.",
        }
    });
  }

  return (
    <div style={{padding: '40px'}}>
```

```
<h1>
  Gift Card Page
</h1>
<h2>
  Customer: {giftCard.firstName} {giftCard.lastName}
</h2>
<h3>
  {giftCard.text}
</h3>
<p>
  {giftCard.instructions}
</p>
{
  giftCard.valid && (
```

Here is the output from the solution code for the **`App.js`** file.



Step-by-step solution

Step 1
You opened the **`App.js`** file and located the **`spendGiftCard()`** function.

Inside the **`spendGiftCard()`** function, you invoked the **`setGiftCard()`** state-updating function, without passing it any parameters or doing anything else with it.

```
function spendGiftCard() {

setGiftCard()

}
```

Step 2
Inside the **`setGiftCard()`** function invocation's parentheses, you passed in an arrow function.

This arrow function has a single parameter, named **`prevState`**. After the arrow, you added a block of code.

```
function spendGiftCard() {

setGiftCard(prevState => {


  })

}
```

Step 3
Next, you returned a copy of the **`prevState`** objectusing the rest operator.

```
function spendGiftCard() {

setGiftCard(prevState => {

return ...prevState

  })

}
```

Step 4
Next, you combined this copy of the **`prevState`** object with those properties that you wanted updated, by updating some of the key-value pairs that already exist on the state object that were initially passed to the **`useState()`** function call.

```
function spendGiftCard() {

setGiftCard(prevState => {

return {

      ...prevState,

text: "Your coupon has been used.",

    }
```

```
  })
}
```

Step 5
Finally, you updated the remaining properties on the state object.

You updated the valid key's value to **`false`**.

Then, updated the instructions key's value to Please visit our restaurant to renew your gift card.

```
function spendGiftCard() {
setGiftCard(prevState => {
return {
        ...prevState,
text: "Your coupon has been used.",
valid: false,
instructions: "Please visit our restaurant to renew your gift card.",
      }
  });
}
```

What is the useEffect hook?

You have been introduced to the primary usage of the **`useEffect`** hook, a built-in React hook best suited to perform side effects in your React components.

In this reading you will be introduced to the correct usage of the dependency array and the different **`useEffect`** calls that can be used to separate different concerns. You will also learn how you can clean up resources and free up memory in your **`useEffect`** logic by returning a function.

The code you place inside the **`useEffect`** hook always runs after your component mounts or, in other words, after React has updated the DOM.

In addition, depending on your configuration via the dependencies array, your effects can also run when certain state variables or props change.

By default, if no second argument is provided to the **`useEffect`** function, the effect will run after every render.

```
useEffect(() => {
   document.title = 'Little Lemon';
 });
```

However, that may cause performance issues, especially if your side effects are computationally intensive. A way to instruct React to skip applying an effect is passing an array as a second parameter to **`useEffect`**.

In the below example, the integer variable **`version`** is passed as the second parameter. That means that the effect will only be re-run if the **`version`** number changes between renders.

```
useEffect(() => {
  document.title = `Little Lemon, v${version}`;
}, [version]); // Only re-run the effect if version changes
```

If **`version`** is 2 and the component re-renders and version still equals 2, React will compare `[2]` from the previous render and `[2]` from the next render. Since all items inside the array are the same, React would skip running the effect.

Use multiple Effects to Separate Concerns

React doesn't limit you in the number of effects your component can have. In fact, it encourages you to group related logic together in the same effect and break up unrelated logic into different effects.

```
function MenuPage(props) {
  const [data, setData] = useState([]);

  useEffect(() => {
    document.title = 'Little Lemon';
  }, []);

  useEffect(() => {
    fetch(`https://littlelemon/menu/${id}`)
      .then(response => response.json())
      .then(json => setData(json));
  }, [props.id]);

  // ...
}
```

Multiple hooks allow you to split the code based on what it is doing, improving code readability and modularity.

Effects with Cleanup

Some side effects may need to clean up resources or memory that is not required anymore, avoiding any memory leaks that could slow down your applications.

For example, you may want to set up a subscription to an external data source. In that scenario, it is vital to perform a cleanup after the effect finishes its execution.

How can you achieve that? In line with the previous point of splitting the code based on what it is doing, the **`useEffect`** hook has been designed to keep the code for adding and removing a subscription together, since it's tightly related.

If your effect returns a function, React will run it when it's time to clean up resources and free unused memory.

```
function LittleLemonChat(props) {
  const [status, chatStatus] = useState('offline');

  useEffect(() => {
    LemonChat.subscribeToMessages(props.chatId, () => setStatus('online'))

    return () => {
      setStatus('offline');
      LemonChat.unsubscribeFromMessages(props.chatId);
    };
  }, []);

  // ...
}
```

Returning a function is optional and it's the mechanism React provides in case you need to perform additional cleanup in your components.

React will make sure to run the cleanup logic when it's needed. The execution will always happen when the component unmounts. However, in effects that run after every render and not just once, React will also clean up the effect from the previous render before running the new effect next time.

Conclusion

In this lesson, you learned some practical tips for using the built-in Effect hook. In particular, you were presented with how to use the dependency array properly, how to separate different concerns in different effects, and finally how to clean up unused resources by returning an optional function inside the effect.

Additional resources

Below is a list of additional resources as you continue to explore React hooks and custom hooks.

In particular, to complement your learning in the “Getting started with hooks” lesson, you can work through the following:

- The article on destructuring assignment describes how the destructuring assignment, which allows you to get values out of the array that gets returned when the `useState` hook is invoked, works in more detail.
- The read props inside the child component link on the Beta version of React docs discusses how to use destructuring assignment to get values out of the props object.
- The useState reference on official React docs website helps you understand how to work with this hook and some of the caveats involved.
- The useEffect reference on official React docs website helps you understand the syntax of this hook and goes into some depth to explain how to use and troubleshoot the `useEffect` hook.

Data fetching using hooks

You learned more about fetching data using hooks and that fetching data from a third-party API is considered a side-effect that requires the use of the `useEffect` hook to deal with the Fetch API calls in React.

You also explored how the response from fetching third-party data might fail, or be delayed, and that it can be useful to provide different renders, based on whether or not the data has been received.

In this reading, you will explore the different approaches to setting up the `useEffect` hook when fetching JSON data from the web. You will also learn why it can be useful to provide different renders, based on whether or not the data has been received.

You have previously learned about using the Fetch API to get some JSON data from a third-party website in plain JavaScript.

You'll be glad to learn that data fetching is not that different in React.

There is only one more ingredient that you need to keep in mind when working with React, namely, that fetching data from a third-party API is considered a side-effect.

Being a side-effect, you need to use the useEffect hook to deal with using the Fetch API calls in React.

To understand what that entails, let me illustrate it with a code example where a component is fetching some data from an external API to display information about a cryptocurrency.

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

export default function App() {
  const [btcData, setBtcData] = useState({});
  useEffect(() => {
    fetch(`https://api.coindesk.com/v1/bpi/currentprice.json`)
      .then((response) => response.json())
      .then((jsonData) => setBtcData(jsonData.bpi.USD))
      .catch((error) => console.log(error));
  }, []);

  return (
    <>
      <h1>Current BTC/USD data</h1>
      <p>Code: {btcData.code}</p>
      <p>Symbol: {btcData.symbol}</p>
      <p>Rate: {btcData.rate}</p>
      <p>Description: {btcData.description}</p>
      <p>Rate Float: {btcData.rate_float}</p>
    </>
  );
}
```

This example shows that in order to fetch data from a third party API, you need to pass an anonymous function as a call to the **`useEffect`** hook.

```
useEffect(
    () => {
        // ... data fetching code goes here
    },
    []
);
```

The code above emphasizes the fact that the **`useEffect`** hook takes two arguments, and that the first argument holds the anonymous function, which, inside its body, holds the data fetching code.

Alternatively, you might extract this anonymous function into a separate function expression or function declaration, and then just reference it.

Using the above example, that code could be presented as follows:

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

export default function App() {
  const [btcData, setBtcData] = useState({});

  const fetchData = () => {
```

```
    fetch(`https://api.coindesk.com/v1/bpi/currentprice.json`)
      .then((response) => response.json())
      .then((jsonData) => setBtcData(jsonData.bpi.USD))
      .catch((error) => console.log(error));
  };

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

  return (
    <>
      <h1>Current BTC/USD data</h1>
      <p>Code: {btcData.code}</p>
      <p>Symbol: {btcData.symbol}</p>
      <p>Rate: {btcData.rate}</p>
      <p>Description: {btcData.description}</p>
      <p>Rate Float: {btcData.rate_float}</p>
    </>
  );
}
```

The code essentially does the same thing, but this second example is cleaner and better organized.

One additional thing that can be discussed here is the **`return`** statement of the above example.

Very often, the response from fetching third-party data might fail, or be delayed. That's why it can be useful to provide different renders, based on whether or not the data has been received.

The simplest conditional rendering might involve setting up two renders, based on whether or not the data has been successfully fetched.

For example:

```
 return someStateVariable.length > 0 ? (
    <div>
      <h1>Data returned:</h1>
      <h2>{someStateVariable.results[0].price}</h2>
    </div>
  ) : (
    <h1>Data pending...</h1>
  );
```

In this example, I'm conditionally returning an **`h1`** and **`h2`**, **if** the length of the **`someStateVariable`** binding's length is longer than 0.

This approach would work if the **`someStateVariable`** holds an array.

If the **`someStateVariable`** is initialized as an empty array, passed to the call to the **`useState`** hook, then it would be possible to update this state variable with an array item that might get returned from a **`fetch()`** call to a third-party JSON data provider.

If this works out as described above, the length of the **`someStateVariable`** would increase from the starting length of zero - because an empty array's length is zero.

Let's inspect the conditional **`return`** again:

```
return someStateVariable.length > 0 ? (
   <div>
     <h1>Data returned:</h1>
     <h2>{someStateVariable.results[0].price}</h2>
   </div>
 ) : (
   <h1>Data pending...</h1>
 );
```

If the data fetching fails, the text of "Data pending..." will render on the screen, since the length of the **`someStateVariable`** will remain being zero.

Conclusion

You learned more about fetching data using hooks and that fetching data from a third-party API is considered a side-effect that requires the use of the **`useEffect`** hook to deal with the Fetch API calls in React.

You also explored how the response from fetching third-party data might fail, or be delayed, and that it can be useful to provide different renders, based on whether or not the data has been received.

Exercise: Can you fetch data?

Open Lab

Instructions

Task

You've learned how to fetch data in React.

In this code lab, you'll practice fetching some data from the *randomuser.me* website's API.

This code lab's app, once completed, is supposed to show a single customer's data for the Little Lemon Restaurant.

The starter code shows only an **`h1`** heading, with the following text: "Data pending..."

Your task is to complete the data fetching using the **`fetch()`** function and to handle the returned Promise object using the **`then()`** methods.

In the **`return`** statement of the App component, you also need to add an **`h2`** heading showing the customer's name and the customer's image from the data fetched from the random user API.

Here's an example screenshot of the completed app served in the browser.

In other words, the completed app should display the following:

1. An **`h1`** heading with the text that reads "Customer data".
2. An **`h2`** heading with the text that reads, for example: "Name: Liam".
3. An **`image`** tag showing an image returned from the fetched data.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

Steps

Step 1

Open the **`App.js`** file.

The **`App.js`** starting code is as follows:

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

```
function App() {
  const [user, setUser] = React.useState([]);

  const fetchData = () => {

  };

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

  return Object.keys(user).length > 0 ? (
    <div>
      <h1>Customer data</h1>

    </div>
  ) : (
    <h1>Data pending...</h1>
  );
}

export default App;
```

Locate the **`fetchData()`** function.

```
const fetchData = () => {

};
```

Inside the **`fetchData()`** function's code block, execute the **`fetch()`** function, passing it a single string argument: **`"https://randomuser.me/api/?results=1"`**.

Step 2
Still inside the **`fetchData()`** function, under the **`fetch()`** function call, add the following piece of code:

```
1
```

```
.then((response) => response.json())
```

Step 3
Continuing from the previous step, add another **`then()`** call, which takes an arrow function.

The passed-in arrow function should receive a **`data`** argument and using that **`data`** argument, it should invoke the **`setUser()`** function, with the **`data`** passed to it.

Step 4
In the return statement of the App component, the starting code is as follows:

```
return Object.keys(user).length > 0 ? (
    <div style={{padding: "40px"}}>
        <h1>Customer data</h1>


    </div>
    ) : (
    <h1>Data pending...</h1>
);
```

Under the **`h1`** heading, you need to add one line of code:

- an **`h2`** heading, with the following code inside:
  `Name:{user.results[0].name.first}`

Step 5
In Step 4, you updated the **`return`** statement of the App component.

In this step, you need to add another line of code under the newly-added **`h2`**.

You need to add an **`img`** element, with the **`src`** attribute holding the following code:

- **`{user.results[0].picture.large}`**

Additionally, you need to add an **`alt`** attribute, as follows:

```
alt=""
```

Remember to self-close the **`img`** tag.

Conclusion

Save the changes and run the app.

Verify that the completed app, behaves as follows:

1. It shows a heading that reads: "Customer data".
2. It shows a subheading, that shows a user name. For example, "Name: Ann".
3. It shows an image of that user under the name.

Solution: Can you fetch data?

Here is the completed solution code for the **`App.js`** file:

```
import React from "react";


function App() {
  const [user, setUser] = React.useState([]);


  const fetchData = () => {
    fetch("https://randomuser.me/api/?results=1")
      .then((response) => response.json())
      .then((data) => setUser(data));
  };


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


  return Object.keys(user).length > 0 ? (
    <div style={{padding: "40px"}}>
      <h1>Customer data</h1>
```

```
      <h2>Name: {user.results[0].name.first}</h2>
      <img src={user.results[0].picture.large} alt="" />
    </div>
  ) : (
    <h1>Data pending...</h1>
  );
}

export default App;
```

Here is the sample output from the solution code for the **`App.js`** file. Note that the lab produces a different random image and name with each refresh, so your output is likely to have a different name and a different customer image than the one shown below.

Important note

If your browser doesn't automatically reload with the updates, please remember to manually refresh your browser to view the changes after modifying **App.js**.

Step-by-step solution

Step 1
Inside the **`fetchData()`** function's code block, you executed the **`fetch()`** function, passing it a single string argument: **`"https://randomuser.me/api/?results=1"`**.

```
const fetchData = () => {
fetch("https://randomuser.me/api/?results=1")
};
```

Step 2
Next, inside the **`fetchData()`** function, under the **`fetch()`** function call, you added the following piece of code:

```
const fetchData = () => {

fetch("https://randomuser.me/api/?results=1")

        .then((response) => response.json())

};
```

Step 3
You then added another **`then()`** call, which takes an arrow function.

The passed-in arrow function, receives a **`data`** argument, and using that **`data`** argument, it invokes the **`setUser()`** function, with the data passed to it.

```
const fetchData = () => {

fetch("https://randomuser.me/api/?results=1")

      .then((response) => response.json())

      .then((data) => setUser(data));

  };
```

Step 4
In the **`return`** statement of the App component, under the **`h1`** heading that reads "Customer data", you added an **`h2`** heading, with the following code: **`Name: {user.results[0].name.first}`**

```
return Object.keys(user).length > 0 ? (

<div style={{padding: "40px"}}>

<h1>Customer data</h1>

<h2>Name: {user.results[0].name.first}</h2>

</div>

    ) : (

<h1>Data pending...</h1>

);
```

Step 5
You then updated the **`return`** statement of the App component by adding another line of code under the newly-added **`h2`**.

You added an **`img`** element, with the **`src`** attribute and an **`alt`** attribute holding the following code: **`{user.results[0].picture.large}`**

```
return Object.keys(user).length > 0 ? (

<div style={{padding: "40px"}}>

<h1>Customer data</h1>

<h2>Name: {user.results[0].name.first}</h2>

<img src={user.results[0].picture.large}alt="" />

</div>

    ) : (

<h1>Data pending...</h1>

);
```

Additional resources

Below is a list of additional resources as you continue to explore React hooks and custom hooks.

In particular, to complement your learning on the 'Rules of hooks and fetching data with hooks' lesson, you can work through the following:

- The Rules of Hooks reading on Reactjs.org website gives you an overview of how to work with the hooks as recommended by the React Core team at Meta.
- The Fetching data with Effects article on React docs discusses fetching data using a few different approaches, including using async / await syntax.
- How to use promises is a resource that describes the "behind-the-scenes" of how data fetching works in greater depth.
- async function is a resource on MDN that discusses the use of the async and await keywords as a more recent way to handle API requests in JavaScript.

When to choose useReducer vs useState

The `useState` hook is best used on less complex data.

While it's possible to use any kind of a data structure when working with `useState`, it's better to use it with primitive data types, such as strings, numbers, or booleans.

The `useReducer` hook is best used on more complex data, specifically, arrays or objects.

While this rule is simple enough, there are situations where you might be working with a simple object and still decide to use the `useState` hook.

Such a decision might stem from the simple fact that working with `useState` can sometimes feel easier than thinking about how the state is controlled when working with `useReducer`.

It might help conceptualizing this dilemma as a gradual scale, on the left side of which, there is the `useState` hook with primitive data types and simple use cases, such as toggling a variable on or off.

At the end of this spectrum, there is the `useReducer` hook used to control state of large state-holding objects.

There's no clear-cut point on this spectrum, at which point you would decide: "If my state object has three or more properties, I'll use the `useReducer` hook".

Sometimes such a statement might make sense, and other times it might not.

What's important to remember is to keep your code simple to understand, collaborate on, contribute to, and build from.

One negative characteristic of `useState` is that it often gets hard to maintain as the state gets more complex.

On the flip side, a negative characteristic of `useReducer` is that it requires more prep work to begin with. There's more setup involved. However, once this setup is completed, it gets easier to extend the code based on new requirements.

Conclusion

You learned about the decision-making process when choosing between `useReducer` and `useState` for working with different types of data.

Custom hooks

React has some built-in hooks, such as the **`useState`** hook, or the **`useRef`** hook, which you learned about earlier. However, as a React developer, you can write your own hooks. So, why would you want to write a custom hook?

In essence, hooks give you a repeatable, streamlined way to deal with specific requirements in your React apps. For example, the **`useState`** hook gives us a reliable way to deal with state updates in React components.

A custom hook is simply a way to extract a piece of functionality that you can use again and again. Put differently, you can code a custom hook when you want to avoid duplication or when you do not want to build a piece of functionality from scratch across multiple React projects. By coding a custom hook, you can create a reliable and streamlined way to reuse a piece of functionality in your React apps.

To understand how this works, let's explore how to build a custom hook. To put this in context, let's also code a very simple React app.

The entire React app is inside the App component below:

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


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


  function increment() {
    setCount(prevCount => prevCount + 1)
  }


  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Plus 1</button>
    </div>
  );
}
```

```
export default App;
```

This is a simple app with an **`h1`** heading that shows the value of the count state variable and a button with an **`onClick`** event-handling attribute which, when triggered, invokes the **`increment()`** function.

The hook will be simple too. It will console log a variable's value whenever it gets updated.

Remember that the proper way to handle **`console.log()`** invocations is to use the **`useEffect`** hook.

So, this means that my custom hook will:

1. Need to use the **`useEffect`** hook and
2. Be a separate file that you'll then use in the App component.

How to name a custom hook

A custom hook needs to have a name that begins with use.

Because the hook in this example will be used to log values to the console, let's name the hook **`useConsoleLog`**.

Coding a custom hook

Now's the time to explore how to code the custom hook.

First, you'll add it as a separate file, which you can name **`useConsoleLog.js`**, and add it to the root of the **`src`** folder, in the same place where the App.js component is located.

Here's the code of the useConsoleLog.js file:

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


function useConsoleLog(varName) {
  useEffect(() => {
    console.log(varName);
  }, [varName]);
}


export default useConsoleLog;
```

Using a custom hook

Now that the custom hook has been coded, you can use it in any component in your app.

Since the app in the example only has a single component, named App, you can use it to update this component.

The **`useConsoleLog`** hook can be imported as follows:

```
import useConsoleLog from "./useConsoleLog";
```

And then, to use it, under the state-setting code, I'll just add the following line of code:

```
useConsoleLog(count);
```

Here's the completed code of the App.js file:

```
import { useState } from "react";
import useConsoleLog from "./useConsoleLog";

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

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

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Plus 1</button>
    </div>
  );
}
```

```
export default App;
```

This update confirms the statement made at the beginning of this reading, which is that custom hooks are a way to extract functionality that can then be reused throughout your React apps

Conclusion

You have learned how to name, build and use custom hooks in React.

Exercise: Create your own custom hook, usePrevious

Instructions

Task

You've learned how to code a custom hook in React. In this code lab, you'll practice writing a brand new custom hook.

The starter code for this app contains code that will display the day of the week within an **`h1`** heading, for example: **`"Today is: Monday"`**. There is also a button under this heading, that reads **`"Get next day"`**.

When a user clicks on this button, the **`h1`** heading updates so that the message shows the next day in the sequence.

Your task is to complete the custom hook named **`usePrevious`** so that the **`h1`** heading shows both the current day and the previous current day before the update.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

Note: Use **`npm start`** to get the starting version of the app running in the browser.

Steps

Step 1
Open the **`App.js`** file and locate the **`usePrevious()`** function as in the below code snippet.

```
function usePrevious(val) {


}
```

Inside the **`usePrevious()`** function's code block, invoke the **`useRef`** hook without any arguments, and assign this invocation to a variable named **`ref`**, declared using the **`const`** keyword.

Step 2

Next, inside the **`usePrevious()`** function declaration, add a call to the **`useEffect()`** hook.

Step 3

Now, pass two parameters, as an argument to the **`useEffect()`** call.

The first parameter should be an arrow function that doesn't accept and arguments. Inside the arrow function's body, assign the **`val`** value to the **`current`** property on the **`ref`** object.

The second parameter is the dependencies array. The dependencies array should list a single variable to be tracked - namely, the **`val`** variable.

Step 4

Add one more line to the body of the **`usePrevious()`** function declaration, specifying the return value of this function.

The **`usePrevious()`** function should return the **`ref.current`** value.

Conclusion

Save the changes and run the app.

Use the checklist below to verify that the completed app is behaving as required:

1. It shows a heading that reads: "Today is: Monday"
2. When the user clicks the "Get next day" button, this updates the heading, which now spans two lines.
3. The first line of the updated heading now reads "Today is: Tuesday", and the second line reads "Previous work day was: Monday".

Solution: Create your own custom hook, usePrevious

Your task was to complete the custom hook named **`usePrevious`** so that the **`h1`** heading shows both the current day and the previous current day before the update.

Here is the completed solution code for the **`App.js`** file:

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

export default function App() {
  const [day, setDay] = useState("Monday");
  const prevDay = usePrevious(day);
  const getNextDay = () => {
    if (day === "Monday") {
      setDay("Tuesday")
    } else if (day === "Tuesday") {
      setDay("Wednesday")
    } else if (day === "Wednesday") {
      setDay("Thursday")
    } else if (day === "Thursday") {
      setDay("Friday")
    } else if (day === "Friday") {
      setDay("Monday")
    }
  }
  return (
    <div style={{padding: "40px"}}>
      <h1>
        Today is: {day}<br />
        {
```

```
          prevDay && (
            <span>Previous work day was: {prevDay}</span>
          )
        }
      </h1>
      <button onClick={getNextDay}>
        Get next day
      </button>
    </div>
  );
}
function usePrevious(val) {
  const ref = useRef();
  useEffect(() => {
    ref.current = val;
  }, [val]);
  return ref.current;
}
```

Steps

Step 1
You should have located the **`usePrevious()`** function.

```
function usePrevious(val) {


}
```

Inside the **`usePrevious()`** function's code block, you needed to invoke the **`useRef`** hook without any arguments, and assign this invocation to a variable named **`ref`**, declared using the **`const`** keyword.

```
function usePrevious(val) {

const ref = useRef();

}
```

Step 2
Next, inside the **`usePrevious()`** function declaration, you needed to add a call to the **`useEffect()`** hook.

```
function usePrevious(val) {

const ref = useRef();

useEffect();

}
```

Step 3
Then, you needed to pass two parameters as argumentsto the **`useEffect()`** function call.

The first parameter should have been an arrow function, without any arguments. Inside the arrow function's body, you should have assigned the **`val`** value to the current property on the **`ref`** object.

The second parameter needed to be the dependencies array. The dependencies array needed to list a single variable - namely, the **`val`** variable.

```
function usePrevious(val) {

const ref = useRef();

useEffect(() => {

ref.current = val;

    }, [val]);

}
```

Step 4
You needed to add one more line to the body of the **`usePrevious()`** function declaration, to specify the **`return`** value of that function.

The **`usePrevious()`** function should have returned the **`ref.current`** value, as follows:

```
function usePrevious(val) {

const ref = useRef();

useEffect(() => {
```

```
ref.current = val;

    }, [val]);

return ref.current;

}
```

Additional resources

Below is a list of additional readings as you continue to explore 'React hooks and custom hooks'.

In particular, to complement your learning in the 'Advanced hooks' lesson, you can work through the following:

- The useReducer hook reference in the React docs discusses the basics of **`useReducer`**, along with specifying initial state and lazy initialization.
- The React docs also has a reference on using the useRef hook which is a great example of various options that are available when working with the **`useRef`** hook.
- The Reusing Logic with Custom Hooks reference in the React docs discusses the dynamics of custom hooks and provides a few practical examples to complement the theory behind them.

WEEKS 3

Types of Children

In JSX expressions, the content between an opening and closing tag is passed as a unique prop called children. There are several ways to pass children, such as rendering string literals or using JSX elements and JavaScript expressions. It is also essential to understand the types of JavaScript values that are ignored as children and don't render anything. Let's explore these in a bit more detail:

String literals

String literals refer to simple JavaScript strings. They can be put between the opening and closing tags, and the `children` prop will be that string.

```
<MyComponent>Little Lemon</MyComponent>
```

In the above example, the `children` prop in `MyComponent` will be simply the string "Little Lemon".

There are also some rules JSX follows regarding whitespaces and blank lines you need to bear in mind, so that you understand what to expect on your screen when those edge cases occur.

1. JSX removes whitespaces at the beginning and end of a line, as well as blank lines:

```
<div>    Little Lemon   </div>

 <div>

   Little Lemon

 </div>
```

2. New lines adjacent to tags are removed:

```
<div>


  Little Lemon

</div>
```

3. JSX condenses new lines that happen in the middle of string literals into a single space:

```
<div>
  Little
  Lemon
</div>
```

That means that all the instances above render the same thing.

JSX Elements

You can provide JSX elements as children to display nested components:

```
<Alert>
  <Title />
  <Body />
</Alert>
```

JSX also enables mixing and matching different types of children, like a combination of string literals and JSX elements:

```
<Alert>
  <div>Are you sure?</div>
  <Body />
</Alert>
```

A React component can also return a bunch of elements without wrapping them in an extra tag. For that, you can use React Fragments either using the explicit component imported from React or empty tags, which is a shorter syntax for a fragment. A React Fragment component lets you group a list of children without adding extra nodes to the DOM. You can learn more about fragments in the additional resources unit from this lesson.

The two code examples below are equivalent, and it's up to your personal preference what to choose, depending on whether you prefer explicitness or a shorter syntax:

```
return (
  <React.Fragment>
```

```
    <li>Pizza margarita</li>
    <li>Pizza diavola</li>
  </React.Fragment>
);

return (
  <>
    <li>Pizza margarita</li>
    <li>Pizza diavola</li>
  </>
);
```

JavaScript Expressions

You can pass any JavaScript expression as children by enclosing it within curly braces, `{}`. The below expressions are identical:

```
<MyComponent>Little Lemon</MyComponent>

<MyComponent>{'Little Lemon'}</MyComponent>
```

This example is just for illustration purposes. When dealing with string literals as children, the first expression is preferred.

Earlier in the course, you learned about lists. JavaScript expressions can be helpful when rendering a list of JSX elements of arbitrary length:

```
function Dessert(props) {
  return <li>{props.title}</li>;
}

function List() {
  const desserts = ['tiramisu', 'ice cream', 'cake'];
  return (
    <ul>
```

```
      {desserts.map((dessert) => <Item key={dessert} title={dessert} />)}
    </ul>
  );
}
```

Also, you can mix JavaScript expressions with other types of children without having to resort to string templates, like in the example below:

```
function Hello(props) {
  return <div>Hello {props.name}!</div>;
}
```

Functions

Suppose you insert a JavaScript expression inside JSX. In that case, React will evaluate it to a string, a React element, or a combination of the two. However, the children prop works just like any other prop, meaning it can be used to pass any type of data, like functions.

Function as children is a React pattern used to abstract shared functionality that you will see in detail in the next lesson.

Booleans, Null and Undefined, are ignored

false, null, undefined, and true are all valid children. They simply don't render anything. The below expressions will all render the same thing:

```
<div />

<div></div>

<div>{false}</div>

<div>{null}</div>

<div>{undefined}</div>

<div>{true}</div>
```

Again, this is all for demonstration purposes so that you know what to expect on your screen when these special values are used in your JSX.

When used in isolation, they don't offer any value. However, boolean values like true and false can be useful to conditionally render React elements, like rendering a Modal component only if the variable `showModal` is true

```
<div>
  {showModal && <Modal />}
</div>
```

However, keep in mind that React still renders some "false" values, like the 0 number. For example, the below code will not behave as you may expect because 0 will be printed when `props.desserts` is an empty array:

```
<div>
  {props.desserts.length &&
    <DessertList desserts={props.desserts} />
  }
</div>
```

To fix this, you need to make sure the expression before `&&` is always boolean:

```
<div>
  {props.desserts.length > 0 &&
    <DessertList desserts={props.desserts} />
  }
</div>
```

```
<div>
  {!!props.desserts.length &&
    <DessertList desserts={props.desserts} />
  }
</div>
```

Conclusion

You have learned about different types of children in JSX, such as how to render string literals as children, how JSX elements and JavaScript expressions can be used as children, and the boolean, null or undefined values that are ignored as children and don’t render anything.

Exercise: Build a Radio Group Component

Instructions

Task

You've learned about some advanced React APIs, **`React.cloneElement`** and **`React.Children.map`** that allow you to modify the children of a component dynamically. In this exercise, you'll use these APIs to build a RadioGroup component.

On HTML, an input of type radio offers a checked property to determine whether a radio button is selected or not.

When building a radio group in React, in other words, a component that represents a set of choices where only one can be selected at a time, an initial implementation might look like this:

```
<RadioOption checked={false} onChange={handleOnChange} value="1" />

<RadioOption checked={true} onChange={handleOnChange} value="2" />

<RadioOption checked={false} onChange={handleOnChange} value="3" />
```

However, this approach is not ideal because it requires the user of the component to keep track of the state of each radio button, as well as setting state change handlers for each option.

Wouldn't it be great to remove any redundancies and reduce the state to the bare minimum while still keeping the functionality intact? Well, this is a great example where component composition shines and enables developers to use a more intuitive and simpler set of props to define a component's API.

Instead of having the user of the component to keep track of the state of each radio button, you can have a parent **`RadioGroup`** component that is aware of the current selection and provides a handler to manage any selection change, without you having to explicitly pass the **`checked`** and **`onChange`** props to each **`RadioOption`** component.

The **`RadioGroup`** component can then leverage the **`children`** prop and use both **`React.cloneElement`** and **`React.Children.map`** to internally pass the **`checked`** and **`onChange`** props to each **`RadioOption`** child.

```
<RadioGroup selected={selectedValue} onChange={handleOnChange}>

  <RadioOption value="1" />

  <RadioOption value="2" />

  <RadioOption value="3" />
```

```
</RadioGroup>
```

In this exercise, you are going to implement this exact component API.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

Steps

Step 1

If you run **npm start** and view the app in the browser you'll notice that the starting React app works as is. The app outputs a simple view with a header and a submit button, but no radio options yet.

Open the **App.js** file. In there you will already see the desired API for the **RadioGroup** and **RadioOption** components. At the moment, they don't render anything on the screen. You don't have to change anything in this file, but just understand the set of props involved in the component design.

Step 2

Open the **Radio/index.js** file. Implement the remaining bits for the **RadioGroup** component. The **RadioOptions** variable is initially set to **null**. Instead, use **React.Children.map** to iterate over the **children** and clone each child using **React.cloneElement**. The result should be assigned to the **RadioOptions** variable.

Each cloned child should receive two additional props, **checked** and **onChange**.

Step 3

Open the **Radio/index.js** file. The **RadioOption** component is incomplete. In particular, it's missing some props in the input element that it renders: **value**, **checked** and **onChange**.

The **RadioOption** component already receives all those props. Your goal is to connect them to the input element.

When adding the **`onChange`** prop to the radio input, which represents the event that gets triggered whenever you interact with it, you can access the **`value`** property of the event target to get the value of the newly selected radio option, as per the code below.

```
const handleChange = (e) => {

  const newValueSelected = e.target.value

}
```

Step 4

Verify that the app works as expected. You should be able to select a radio option and see how the submit button gets enabled as soon as a selection is made.

Tips

The **`RadioGroup`** component receives the **`selected`** prop, a string that represents the value of the currently selected radio option. However, an individual **`RadioOption`** component only cares about whether it is selected or not, via the boolean **`checked`** prop. You would have to perform some small business logic inside the **`RadioGroup`** component to translate the **`selected`** prop to the **`checked`** prop that each **`RadioOption`** child receives.

Conclusion

In this lab, you learned through a practical example how to leverage both **`React.cloneElement`** and **`React.Children.map`** APIs to build modular components that embrace the **`children`** prop. That allows you to offer a simple API to consumers and, as a result, minimize the amount of external local state required to make them work.

Additional resources

Here is a list of additional resources for JSX Deep Dive:

- Chakra-UI is an open-source component library that embraces all the concepts explained during this lesson, being a nice option if you would like to start your project with a set of atomic components that have been carefully designed with flexibility in mind, so that they can be customized as per your theme requirements.
- Compound components with hooks is an article that illustrates how a combination of component composition, context and hooks can lead to a clean and concise component design.
- React Fragments from the official React docs illustrates how to group a list of React children without adding extra nodes to the DOM.

WEEK 3

Higher-order components

In a previous lesson, you learned about Higher-order components (HOC) as a pattern to abstract shared behavior, as well as a basic example of an implementation.

Let's dive deeper to illustrate some of the best practices and caveats regarding HOCs.

These include never mutating a component inside a HOC, passing unrelated props to your wrapped component, and maximizing composability by leveraging the `Component => Component` signature.

Don't mutate the original component

One of the possible temptations is to modify the component that is provided as an argument, or in other words, mutate it. That's because JavaScript allows you to perform such operations, and in some cases, it seems the most straightforward and quickest path. Remember that React promotes immutability in all scenarios. So instead, use composition and turn the HOC into a pure function that does not alter the argument it receives, always returning a new component.

```
const HOC = (WrappedComponent) => {
  // Don't do this and mutate the original component
  WrappedComponent = () => {



  };
 …
}
```

Pass unrelated props through to the Wrapped Component

HOC adds features to a component. In other words, it enhances it. That's why they shouldn't drastically alter their original contract. Instead, the component returned from a HOC is expected to have a similar interface to the wrapped component.

HOCs should spread and pass through all the props that are unrelated to their specific concern, helping ensure that HOCs are as flexible and reusable as possible, as demonstrated in the example below:

```
const withMousePosition = (WrappedComponent) => {

  const injectedProp = {mousePosition: {x: 10, y: 10}};

  return (originalProps) => {

    return <WrappedComponent injectedProp={injectedProp} {...originalProps} />;

  };

};
```

Maximize composability

So far, you have learned that the primary signature of a HOC is a function that accepts a React component and returns a new component.

Sometimes, HOCs can accept additional arguments that act as extra configuration determining the type of enhancement the component receives.

```
const EnhancedComponent = HOC(WrappedComponent, config)
```

The most common signature for HOCs uses a functional programming pattern called "currying" to maximize function composition. This signature is used extensively in React libraries, such as React Redux, which is a popular library for managing state in React applications.

```
const EnhancedComponent = connect(selector, actions)(WrappedComponent);
```

This syntax may seem strange initially, but if you break down what's happening separately, it would be easier to understand.

```
const HOC = connect(selector, actions);

const EnhancedComponent = HOC(WrappedComponent);
```

`connect` is a function that returns a higher-order component, presenting a valuable property for composing several HOCs together.

Single-argument HOCs like the ones you have explored so far, or the one returned by the connect function has the signature `Component => Component`. It turns out that functions whose output type is the same as its input type are really easy to compose together.

```
const enhance = compose(

  // These are both single-argument HOCs

  withMousePosition,

  withURLLocation,

  connect(selector)

);
```

```
// Enhance is a HOC

const EnhancedComponent = enhance(WrappedComponent);
```

Many third-party libraries already provide an implementation of the compose utility function, like lodash, Redux, and Ramda. Its signature is as follows:

`compose(f, g, h)` is the same as `(...args) => f(g(h(...args)))`

Caveats

Higher-order components come with a few caveats that aren't immediately obvious.

1. Don't use HOCs inside other components: always create your enhanced components outside any component scope. Otherwise, if you do so inside the body of other components and a re-render occurs, the enhanced component will be different. That forces React to remount it instead of just updating it. As a result, the component and its children would lose their previous state.

```
const Component = (props) => {

  // This is wrong. Never do this

  const EnhancedComponent = HOC(WrappedComponent);

  return <EnhancedComponent />;

};


// This is the correct way

const EnhancedComponent = HOC(WrappedComponent);

const Component = (props) => {

  return <EnhancedComponent />;

};
```

1. Refs aren't passed through: since React `refs` are not `props`, they are handled specially by React. If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component. To solve this, you can use the React.forwardRef API. You can learn more about this API and its use cases in the additional resources section from this lesson.

Conclusion

And in summary, you have examined higher-order components in more detail. The main takeaways are never mutating a component inside a HOC and passing unrelated props to your wrapped component.

You also learned how to maximize composability by leveraging the `Component => Component` signature and addressed some caveats about HOC.

Exercise: Implementing scroller position with render props

Instructions

Task

You've learned about render props and how it's a viable alternative to higher-order components (HOCs) to encapsulate cross-cutting concerns. In a previous video, you saw a possible implementation of a mouse position tracker using higher-order components. In this exercise, you'll implement the same specifications but using a render prop component instead.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

Steps

Step 1
Open the **`App.js`** file.

Complete the implementation of the **`MousePosition`** component. Specifically, you'll need to:

- Implement the body **`ofhandleMousePositionChangeinsideuseEffect`**
- Implement the return statement of the component.

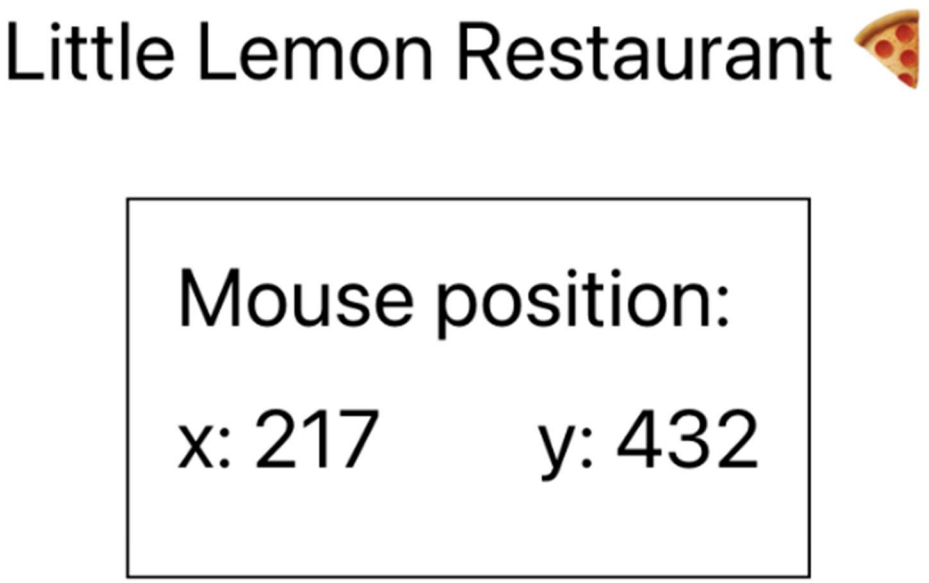
Step 2
Tweak the implementation of **`PanelMouseLogger`**. The requirements are:

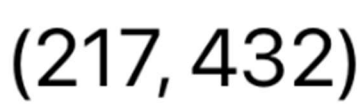
- The component should not receive any props.
- The component should not have any **`if`** statements.
- The component should leverage **`theMousePosition`** render prop to show the coordinates in a panel fashion. The panel UI is already provided to you, your goal is to connect the UI with the mouse position data.

Step 3
Tweak the implementation of **`PointMouseLogger`**. The requirements are:

- The component should not receive any props.
- The component should not have any **`if`** statements.
- The component should leverage **`theMousePosition`** render prop to show the coordinates in a point representation. The point UI is already provided to you, your goal is to connect the UI with the mouse position data

Save all the changes and run the app. Preview the updates in the browser, and confirm that the page shows two distinct interfaces that display the safe information (mouse position) in different fashion, one as a panel and another as a point coordinates.





Solution: Implementing scroller position with render props

Here is the completed solution code for the App.js file:

```
import "./App.css";
import { useEffect, useState } from "react";

const MousePosition = ({ render }) => {
  const [mousePosition, setMousePosition] = useState({
    x: 0,
    y: 0,
  });

  useEffect(() => {
    const handleMousePositionChange = (e) => {
      setMousePosition({
        x: e.clientX,
        y: e.clientY,
      });
    };

    window.addEventListener("mousemove", handleMousePositionChange);

    return () => {
      window.removeEventListener("mousemove", handleMousePositionChange);
    };
  }, []);

  return render({ mousePosition });
};

const PanelMouseLogger = () => {
  return (
    <div className="BasicTracker">
      <p>Mouse position:</p>
      <MousePosition
        render={({ mousePosition }) => (
          <div className="Row">
            <span>x: {mousePosition.x}</span>
            <span>y: {mousePosition.y}</span>
          </div>
```

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

1. Implement the body of `handleMousePositionChange`

The **`mousemove`** handler function receives an event as parameter that contains the mouse coordinates as **`clientX`** and **`clientY`** properties. Therefore you can provide a position update by calling the state setter **`setMousePosition`** with the new values.

```
const handleMousePositionChange = (e) => {
 setMousePosition({
   x: e.clientX,
   y: e.clientY,
 });
};
```

2. Implement the `return` statement of the component

The **`MousePosition`** component receives a **`render`** prop, which is the special prop name designed by convention to specify a function that returns some JSX. Since the **`MousePosition`** component does not take care of any visualization logic, but rather encapsulates cross-cutting concerns, it should return the result of calling the render function with the **`mousePosition`** as an argument. In other words, it's up to the components that consume **`MousePosition`** to specify what sort of UI they want to display when they receive a new value of the mouse position on the screen.

```
return render({ mousePosition })
```

Step 2

The **`PanelMouseLogger`** component should not receive any props. Hence, the early return from the previous implementation if no props were provided is no longer needed.

Instead, the **`mousePosition`** is now injected as the first argument of the render function prop that **`MousePosition`** uses. It's in this render function body where the previous JSX should be extracted and returned.

```
const PanelMouseLogger = () => {
 return (
   <div className="BasicTracker">
     <p>Mouse position:</p>
```

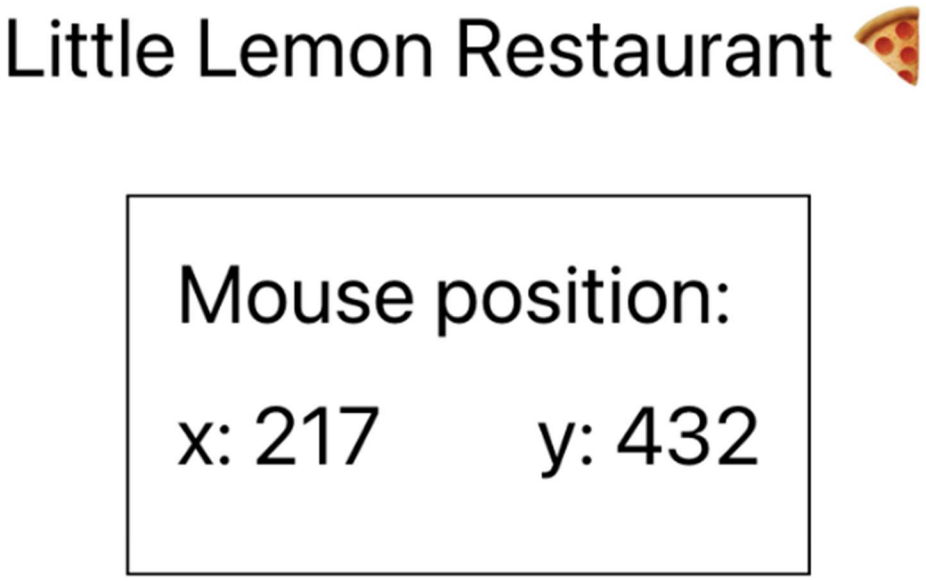
```
      <MousePosition
        render={({ mousePosition }) => (
          <div className="Row">
            <span>x: {mousePosition.x}</span>
            <span>y: {mousePosition.y}</span>
          </div>
        )}
      />
    </div>
  );
};
```

Step 3

Similarly, as in step 2, the component should not receive any props and the early if statement should be removed. The particular UI for this component is provided as part of the render prop as well.

```
const PointMouseLogger = () => {
  return (
    <MousePosition
      render={({ mousePosition }) => (
        <p>
          ({mousePosition.x}, {mousePosition.y})
        </p>
      )}
    />
  );
};
```

At this point, the implementation has been completed and you should see the following result when you run the app in the browser:



Additional resources

Here is a list of additional resources as you continue to explore Reusing Behavior:

- Downshift is a popular open-source library that implements an autocomplete, combo box or select experience using the `render` prop pattern.
- Render props from the official React docs.
- Higher Order Components from the official React docs.
- Forwarding Refs from the official React docs showcases in detail how to forward refs in higher-order components, so that they are passed through properly.

Exercise: Writing more test scenarios

Instructions

Task

You've learned about Jest and react-testing-library to create automated tests for your components. Now it's time to write some of your own tests! Remember the Feedback form Little Lemon put together to gather feedback about their recipes? In a previous lesson, you were introduced to a test scenario where the app verified that users who provided less than a score of 5 could only submit their form if feedback was also provided.

In this exercise, you'll create two more test scenarios to verify the form submission works as expected:

1. User is able to submit the form if the score is lower than 5 and additional feedback is provided.
2. User is able to submit the form if the score is higher than 5, without additional feedback.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

Steps

Step 1

Open the `App.test.js` file. There you'll encounter two new test scenarios that should be completed.

Step 2

After writing the test scenarios, run the tests to verify they pass. For that, execute `npm test` in the terminal.

Tip

Explore the `FeedbackForm` component to understand the JSX it returns and how you can access the elements in the tests. For more information about react-testing-library queries, check out the documentation.

Conclusion

In this exercise, you gained more experience when writing tests for your React Components. You were introduced to different screen query methods to locate DOM elements and learned about new Jest matchers to perform valuable assertions.

Solution: Writing more test scenarios

Here is the completed solution code for the App.test.js file:

```
import { fireEvent, render, screen } from "@testing-library/react";
import FeedbackForm from "./FeedbackForm";

describe("Feedback Form", () => {
  test("User is able to submit the form if the score is lower than 5 and addition
al feedback is provided", () => {
    const score = "3";
    const comment = "The pizza crust was too thick";
    const handleSubmit = jest.fn();
    render(<FeedbackForm onSubmit={handleSubmit} />);

    const rangeInput = screen.getByLabelText(/Score:/);
    fireEvent.change(rangeInput, { target: { value: score } });

    const textArea = screen.getByLabelText(/Comments:/);
    fireEvent.change(textArea, { target: { value: comment } });

    const submitButton = screen.getByRole("button");
    fireEvent.click(submitButton);

    expect(handleSubmit).toHaveBeenCalledWith({
      score,
      comment,
    });
  });

  test("User is able to submit the form if the score is higher than 5, without ad
ditional feedback", () => {
    const score = "9";
    const handleSubmit = jest.fn();
    render(<FeedbackForm onSubmit={handleSubmit} />);

    const rangeInput = screen.getByLabelText(/Score:/);
    fireEvent.change(rangeInput, { target: { value: score } });

    const submitButton = screen.getByRole("button");
    fireEvent.click(submitButton);

    expect(handleSubmit).toHaveBeenCalledWith({
      score,
      comment: ""
```

```
});
```

Steps

Step 1

The first test scenario has the following specification:

User is able to submit the form if the score is lower than 5 and additional feedback is provided

The test scenario already contains some initial code that acts as boilerplate before getting to the bulk of the test, in particular:

- Two variables that hold the desired state of the form, a score of 3 and an additional comment.
- A mock function that is called when submitting the form.
- The rendering of the form component.
- The final assertion that should make the test pass.

If you run as it is, the test will fail stating that the mock function has not been called at all. That is because no interactions have occurred yet and it's your task to write those.

The first user interaction that needs to happen is to set the score as 3. The following code achieves that:

```
const rangeInput = screen.getByLabelText(/Score:/);
fireEvent.change(rangeInput, { target: { value: score } });
```

The first line grabs a reference to the range input component by using the global screen object from react-testing-library and the query **`getByLabelText`** to find a label that contains the exact text **`Score:`**

Then, a change event is simulated on the input, passing as the event an object with the **`value`** property set to the variable **`score`**: **`event.target.value = score`**

After that, a second user interaction is required to set the additional comment. This is the code that accomplishes that:

```
const textArea = screen.getByLabelText(/Comments:/);
fireEvent.change(textArea, { target: { value: comment } });
```

No further explanation is needed regarding those two lines, since they mimic the same interaction as with the range input.

Last but not least, a submission of the form should be simulated by calling the below two lines:

```
const submitButton = screen.getByRole("button");
fireEvent.click(submitButton);
```

In this particular instance, the button is referenced by using a different query on the global screen object, **`getByRole`**. This query looks for an element whose role attribute is set to **`"button"`**, which is inherent in all button HTML tags.

The form is finally submitted via firing a click event on the button instance.

If you run the command **`npm test`** in your terminal, the test should pass now.

Let's now cover the second scenario.

User is able to submit the form if the score is higher than 5, without additional feedback

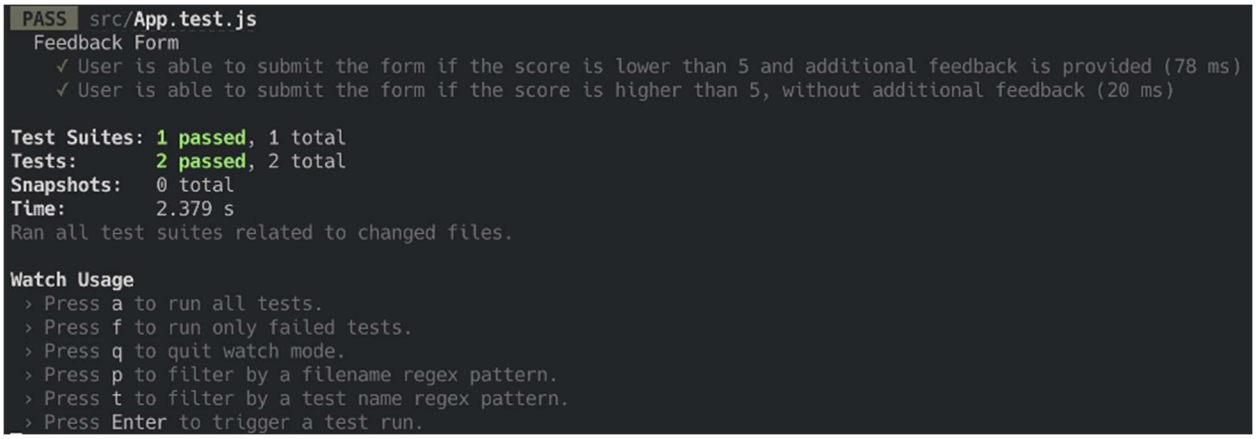
The below represents the code you need to write to make the test pass.

```
const rangeInput = screen.getByLabelText(/Score:/);
fireEvent.change(rangeInput, { target: { value: score } });

const submitButton = screen.getByRole("button");
fireEvent.click(submitButton);
```

This test is simpler and there is nothing new besides skipping any interaction with the text area, since no additional feedback is required when the score provided is higher than 5, being 9 in this scenario.

Step 2

If you run **`npm test`** after adding the lines above, both of your tests should pass successfully, with the terminal providing the below output.



Introduction to continuous integration

Introduction

Continuous Integration (CI) is a software development technique in which developers use a version control system, like Git, and push code changes daily, multiple times a day. Instead of building out features in isolation and integrating them at the end of the development cycle, a more iterative approach is employed.

Each merge triggers an automated set of scripts to automatically build and test your application. These scripts help decrease the chances that you introduce errors in your application.

If some of the scripts fail, the CI system doesn't progress to further stages, issuing a report that developers can use to promptly assess what was wrong and resolve the problem.

This reading will teach you why embracing a CI tool is essential for your software development process. You will also explore a typical development workflow that you can integrate into your CI system and some of the main benefits of using CI.

Why do we need CI?

In new product development, the time to figure everything out up front is limited. Taking smaller steps helps estimate more accurately and validate more often. Having a shorter feedback loop involves more iterations. This number of iterations, not the number of hours invested, drives the process forward.

Working in long feedback loops is risky for software development teams, increasing the chances of introducing errors. Integrating new changes is also time-consuming.

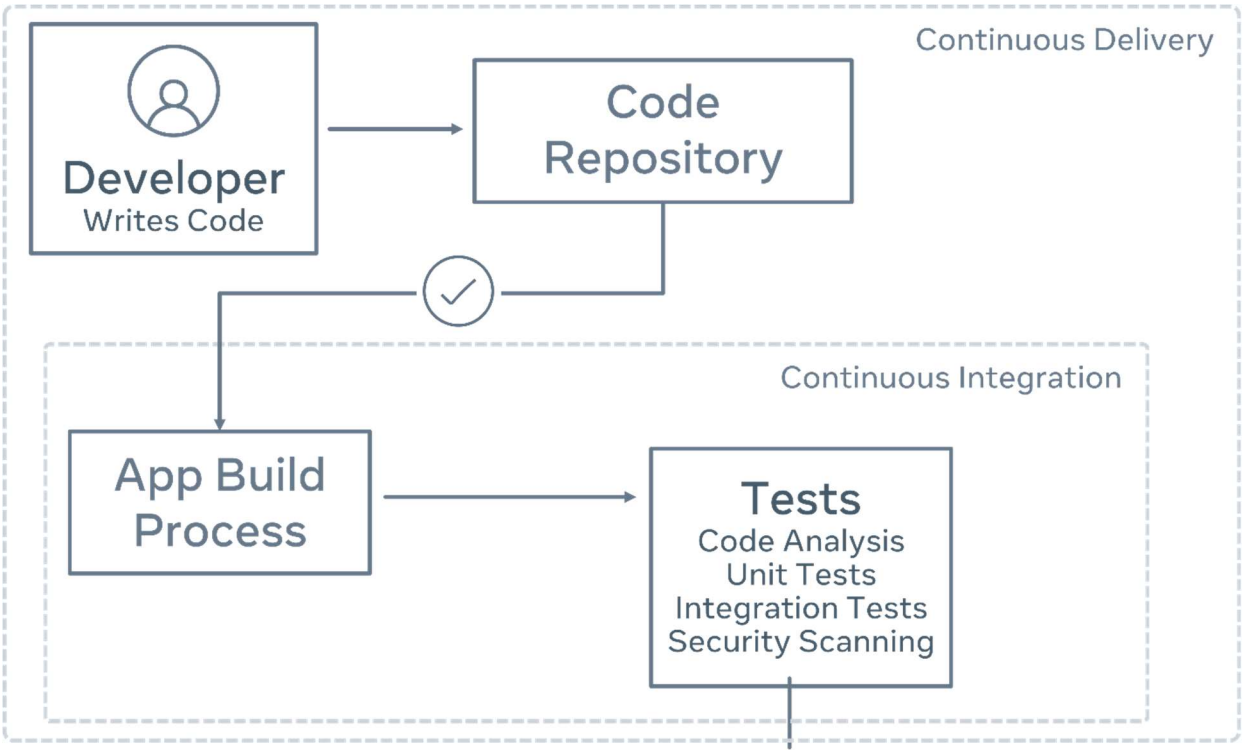
By automating all integration steps and having small controlled changes, developers avoid repetitive work and minimize human errors. The CI tool monitors the central code repository and prevents people from deciding when and how to run tests. Every time there is a new commit, it runs all automated tests.

Based on the outcome, it either accepts the commit if all tests passed successfully or reject it if there was a failure.

CI Pipeline

Below is a graphical representation of a typical CI process as a pipeline. When new code enters one end, a new version of the app gets built automatically, and a suite of automated tests is run against it.

Continuous Integration is a small part of a more significant process called Continuous Delivery. However, that's outside the scope of the purpose of this lesson, and you can check more information in the additional resources section.



A typical development workflow

Here is a simplified CI workflow that companies often embrace daily as part of their software development process:

- A developer from the team creates a new branch of code in Github, performs changes in the code, and commits them.
- When the developer pushes its work to GitHub, the CI system builds the code on its servers and runs the automated test suite.
- Suppose the CI system detects any error in the CI pipeline. In that case, the developer who pushed the code gets a notification, for example, via email, and the status of CI changes to red. The developer is responsible for analyzing what went wrong and fixing the problem.

- Otherwise, if the status is green and all goes well, the pipeline moves to its next stage, which usually involves deploying a new version of the application to a staging server. This new version can be used internally by the Quality Assurance (QA) team to verify the changes in a production-like environment.

Benefits of continuous integration

Some of the benefits for your software development teams are:

- Improved developer productivity: CI frees developers from manual tasks and the pain of integrating their code with other system parts. They can instead focus on programming the logic that delivers the business's desired features.
- Deliver working software more often: CI is a way for your team to build and test every source code change automatically. This fast CI feedback loop delivers more value to customers than teams that rely on manual integrations of each other's work. This foundation enables a software delivery process to be efficient, resilient, fast, and secure.
- Find bugs earlier, and fix them faster: The automated testing process can include different checks, like verifying code correctness, validating application behavior, or making sure the coding style follows industry-standard conventions. A CI tool provides instant feedback to developers on whether the new code they wrote works or introduces bugs or regression in quality. Mistakes that are caught early on are the easiest to fix.

Conclusion

In this reading, you have had an introduction to continuous Integration and why embracing a CI tool is important for your software development process.

You also learned about a typical development workflow that can be integrated into your CI system and explored some of the main benefits of using CI.

Additional resources

Here is a list of additional resources as you continue to explore Integration tests with React Testing Library:

- React testing library official documentation.
- Jest official documentation.
- Continuous delivery is a great article from Atlassian that illustrates the differences between Continuous integration, delivery and deployment, and how they all tie together.
- Practical test pyramid is an extensive article that dives into the importance of test automation, showing you which kind of tests you should be looking for in the different levels of the pyramid and providing practical examples on how those can be implemented.

WEEK 4

About the final project

What is the purpose of the portfolio project?

The primary purpose of an assessment is to check your knowledge and understanding of the key learning objectives of the course you have just completed. Most importantly, assessments help you establish which topics you have mastered and which require further focus before completing the course. Ultimately, the assessment is designed to help you make sure that you can apply what you have learned. This assessment's learning objective is to allow you to create a React application or App.

How do I prepare for the portfolio project?

You will have already encountered exercises, knowledge checks, in-video questions and other assessments as you have progressed through the course.

The final project requires you to complete a portfolio in React. You will be provided with code snippets, and your task is to use these, plus any of your own code to complete your developers' portfolio.

This is a creative project, and the goal is to use as many React concepts as possible within this portfolio. You can use component composition, code reusability, hooks, manage state, interact with an external API, create forms, lists and so on.

You will be provided with code snippets and your task is to use these, plus any of your own code, to complete a portfolio app that contains:

- A header with external links to social media accounts and internal links to other sections of the page.
- A landing section with an avatar picture and a short bio.
- A section to display your featured projects as cards in a grid fashion.
- A contact me section with a form to allow visitors to contact you.

Review the final project

You will take part in a peer review exercise in which you will submit your completed portfolio app for two of your peers to review. You will also be required to review two of your peers' portfolio apps.

When you submit your assignment, other learners in the course will review and grade your work. They will be looking at the Portfolio page functionality based on the following criteria:

- Did the header have external links that take you to different social apps?
- Did the header have internal links that, when clicked, will smoothly scroll into their corresponding section?
- Was the landing section filled with an avatar, name and a short bio?
- Did the project section display a 2x2 grid with each project rendered in a card widget?
- Was the Contact Me form business logic implemented as per the requirements?
- Was the header hidden/shown depending on the scroll direction? Did it happen with a smooth transition animation?
- Can you suggest any improvements for the portfolio app?

You'll also need to give feedback on and grade the assignments of two other learners using the same criteria.

Popular external libraries

In this reading you will learn about some well-designed UI libraries, such as ChakraUI, that can speed up your application delivery.

You will also explore how to simplify your form design with tools like Formik, and write declarative validation rules with chain operators using Yup.

Chakra UI

UI libraries are a great way to speed up the development process. They provide a set of robust, well-tested and highly configurable pre-built components that you can use to create your applications. Those components act as atoms or building blocks, laying the foundation to create more complex components.

One of the most popular UI solutions is Chakra UI. Chakra UI is a simple, modular and accessible component library that provides you with the building blocks you need for your React applications.

Chakra groups its different components by categories, like layout, forms, data display, feedback, typography or overlay.

Layout components are in charge of setting virtual delimiters or boundaries for your content. They also manage how their children are laid (row or column) and the spacing between them among other properties. Some layout components to highlight are:

- HStack and VStack: they display children using flex properties and stack elements horizontally or vertically respectively. Both take a spacing prop that allows you to set the spacing between the elements.
- Box: it allows you to create a box with a background color, border, shadow, etc. It takes a bg prop that allows you to set the background color.

Typography is also an important category that is worth mentioning. There are two main components from this group:

- Heading: renders one of the different DOM header tags (`h1`, `h2`, `h3`…). It takes a size prop that allows you to set the size of the heading and an as prop to specify the particular semantic HTML tag.

```
<Heading as='h2' size='2xl'>
    Little Lemon
 </Heading>
```

- Text: is used to render text and paragraph within an interface. It offers a `fontSize` prop to increase the font size of the text.

```
<Text fontSize='lg'>Best restaurant in town</Text>
```

In order to see all the different component categories and the different props each component accepts, you can check the official documentation page.

Style props

Chakra uses style props to provide css directives directly as props to the different components. You can find a reference of all the available style props in the Chakra UI documentation.

As a general rule, you can consider *camelCase* versions of css styles to be valid style props. But you can also leverage the shorthand version. For example, instead of using `backgroundColor`, you can use `bg`.

```
<Box backgroundColor='tomato' /> is equivalent to <Box bg='tomato' />
```

Putting all together, the below example represents three boxes stacked in a row, with a vertical space of 16px between boxes, where each box has a height of 40px and a different background color, as well as a particular number as its children:

```
<HStack spacing=”16px”>
  <Box h='40px' bg='yellow.200'>
    1
  </Box>
  <Box h='40px' bg='tomato'>
    2
  </Box>
  <Box h='40px' bg='pink.100'>
    3
  </Box>
</HStack>
```

Formik and Yup

Formik is another popular open-source library that helps you to create forms in React. The library takes care of the repetitive tasks of managing the state of the form, validation and submission, so you can focus on the business logic of your application. It does so by providing a set of components and hooks that you can plug into your forms.

Yup is a JavaScript open-source library used to validate the form data before submitting it to the server. It provides a set of chainable operators that you can apply to your form fields to declaratively specify the validation rules.

Formik comes with built-in support for schema based form-level validation through Yup, so they work together seamlessly.

The most important component from Formik is the **`useFormik`** hook. This hook handles all the different states of your form. It only needs a configuration object as an argument.

Let's break down the hook usage with some code example:

```
import * as Yup from 'yup';
import { useFormik } from 'formik';

// The below code would go inside a React component
const {
  values,
  errors,
  touched,
getFieldProps,
handleSubmit,
} = useFormik({
intialValues: {
    comment: "",
  },
onSubmit: (values) => {
    // Handle form submission
  },
validationSchema: Yup.object({
    comment: Yup.string().required("Required"),
  }),
});
```

The **`useFormik`** hook takes an object as an argument with the following properties:

- **`initialValues`**: An object with the initial values of the form fields
- **`onSubmit`**: A function that will be called when the form is submitted, with the form values as an argument. In that example you could access the message via **`values.comment`**.

- **`validationSchema`**: A Yup schema that will be used to validate the form fields. In that example, the message is a field with a string value that is required. As you can see the rules are human-readable and easy to understand.

The hook returns an object with the following properties:

- **`values`**: An object with the current values of the form fields. In that example you could access the message via **`values.comment`**.
- **`errors`**: An object with the current errors of the form fields. If validation fails for the "comment" field, which would be the case if the input has been touched and its value is empty, according to the validation schema, you could access the message error via errors.comment. In that particular case, the message error would be "Required". If the field is valid though, the value will be undefined.
- **`touched`**: An object with the current touched state of the form fields. You can use this to determine if a field has been touched (focused at least once) or not. In that example, you could access the comment state via **`touched.comment`**. If the field has been touched, the value will be true, otherwise it will be false.
- **`getFieldProps`**: A function that takes a field name as an argument and returns an object with the following properties:
  - **`name`**: The field name.
  - **`value`**: The current value of the field.
  - **`onChange`**: The **`handleChange`** function.
  - **`onBlur`**: A function that will be called when the field is blurred. It updates the corresponding field in the touched object.

The way you would use this function is by spreading the returned object into the input element. For example, if you had an input element with the name "comment", you would do something like this:

```
<input {...getFieldProps("comment")} />
```

- **`handleSubmit`**: A function that will be called when the form is submitted. It takes an event as an argument and calls the **`onSubmit`** function with the values object as an argument. You should hook this function to the form **`onSubmit`** event.

Conclusion

In this reading, you have learned about three of the most popular libraries that can save you precious time during your app development. Their main goal is to take care of mundane and repetitive tasks and let you focus on the stuff that matters.

You were introduced to Chakra UI as a way to leverage well designed components that you can put together to build more complex interfaces.

Finally, you gained knowledge about an open-source library called Formik and its perfect companion, Yup, to create complex React forms with ease.

Create your portfolio

Instructions
Task

In this final lab, you are going to create a portfolio page for yourself. You will be using the skills you have learned in this course to create a page that showcases your work. The portfolio page will be a single page that will contain the following sections:

- A header with external links to social media accounts and internal links to other sections of the page
- A landing section with an avatar picture and a short bio
- A section to display your featured projects as cards in a grid fashion
- A contact me section with a form to allow visitors to contact you

Here you will have the opportunity to use some popular open source libraries that will help you to have a more polished and professional looking page.

Before proceeding further, let's provide an introduction to the libraries you will be using.

Libraries

Chakra UI

Chakra UI comes pre-configured with this lab, so you don't have to worry about installing it or setting it up.

The components from this library you will need to use are already imported from the **`@chakra-ui/react`** package at the top of each corresponding file. If you don't see a component already imported, it's because you probably won't need it. In any case, feel free to check their official documentation to see all the components at your disposal and their corresponding props.

Formik and Yup

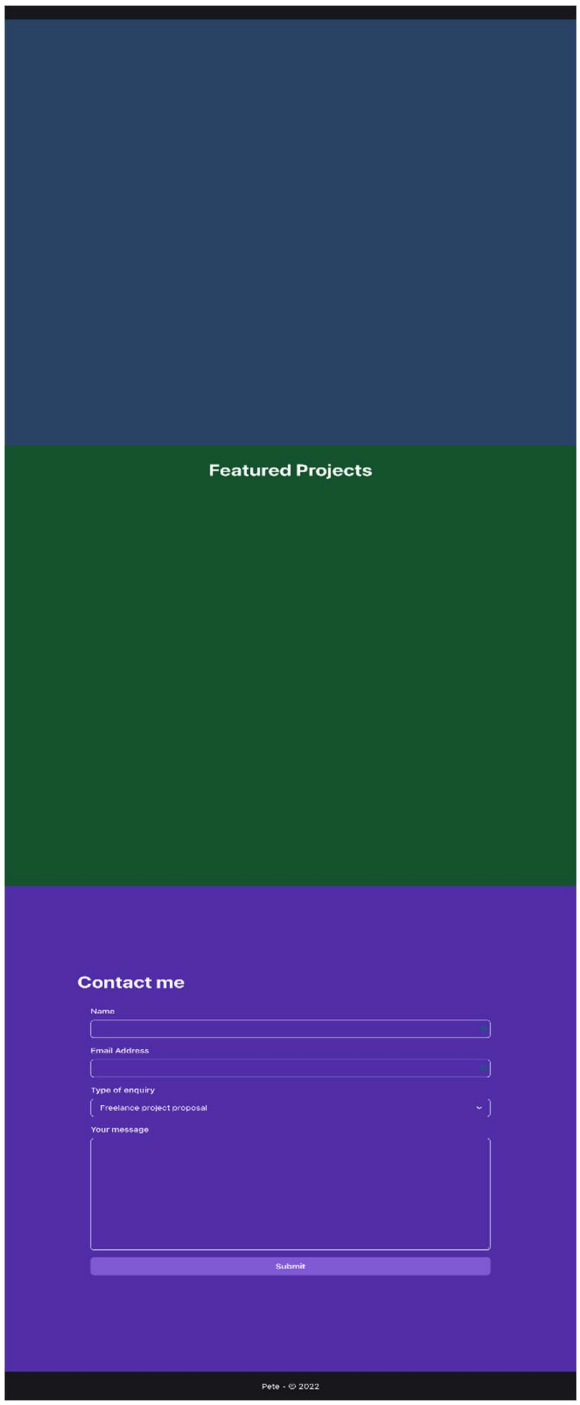
The Formik library is already set up in the project as well, so no extra configuration is needed. In this lab, you will be only using the **`useFormik`** hook from the Formik library, as well as the global Yup object to define the Contact Me form validation rules.

The form UI will be implemented using Chakra UI components.

Note: Before you begin, make sure you understand how to work with the Coursera Code Lab for the Advanced React course.

If you run **`npm start`** and view the app in the browser, you'll notice that the starting React app works as is.

The app outputs a page with an empty header, 3 different full-height sections and a footer. Every section has a different background color. The first 2 sections will be empty and the third section will contain all the UI elements for the Contact Me form.



Steps

Once you open the code lab, you need to install *Chakra UI* and other referenced libraries by running the **`npm install`** command from the built-in terminal in the code lab. To toggle the built-in terminal, you need to click the View menu item, then choose the Terminal in the dropdown.

Once the terminal is open (visible), you can run the **`npm install`** command. This will install all the required missing dependencies, so that you can begin working on the task.

Step 1
Open **`Header.js`** file. You will see a header component with black background, but no content.

a) **Add external social media links to the header on the left side of the page**.

The implementation should be placed inside the first **`nav`** element. The data is already provided in the socials array at the top of the file.

Use the HStack component to stack the links horizontally. Each social should be a tag with a **`href`** attribute pointing to the corresponding social media page. The **`a tag`** should have as children a **`FontAwesomeIcon`** component, which is already imported for you.

The **`FontAwesomeIcon`** component takes 2 props:

- **`icon`**: The icon to be displayed. In this case, you should use the icon prop from the social object.
- **`size`** : The size of the icon. You can use the **`2x`** value.

You can check below an example of how to render it:

```
<FontAwesomeIcon icon="fab" size="2x" />
```

b) **Add internal links to the Projects section and Contact Me section**

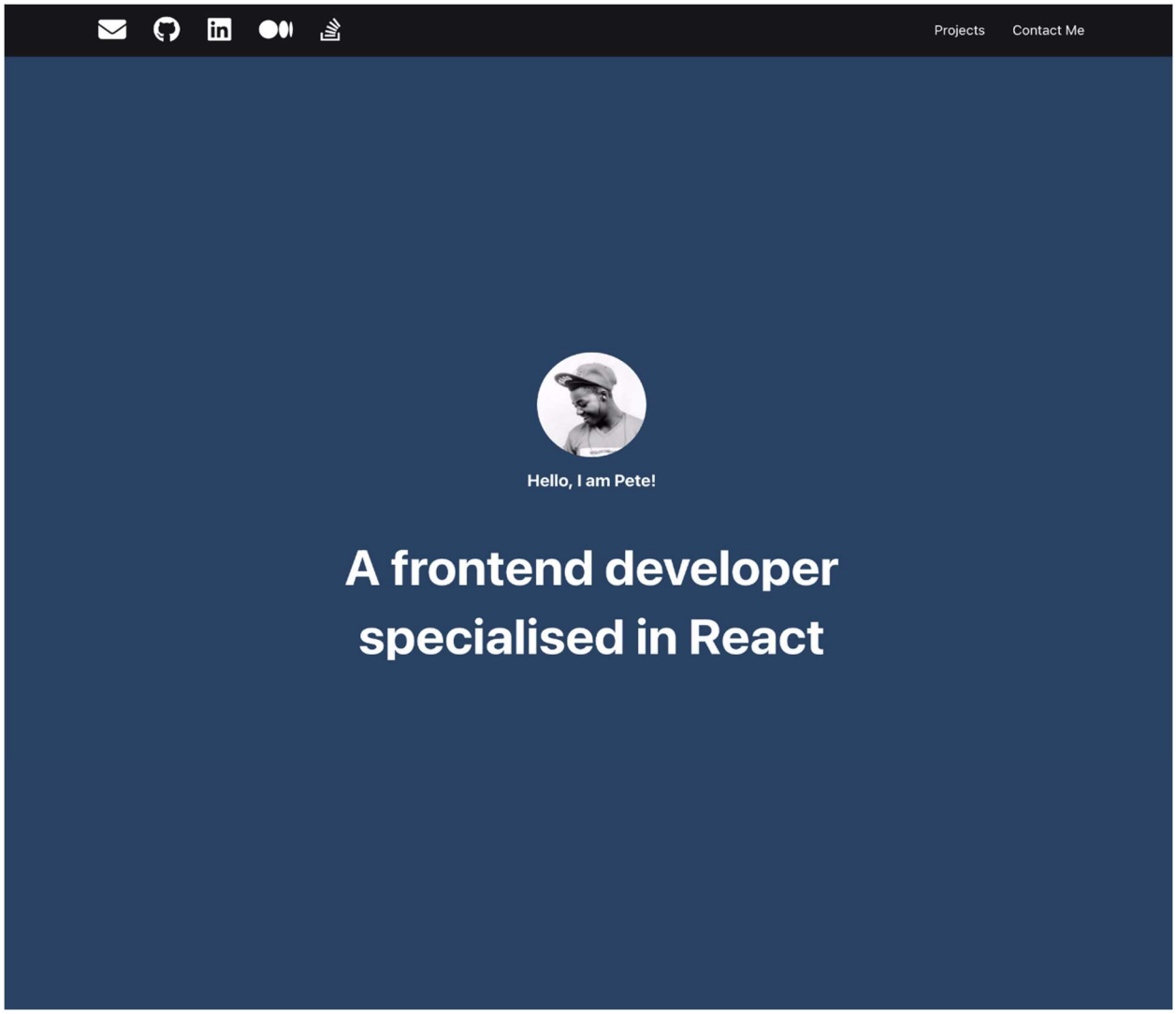
Each link should be an **`a tag`**. Each **`a tag`** should have as children the name of the section: "Contact Me" and "Projects". When clicking on the link, the url should show the corresponding section. For example, when clicking on the "Contact Me" link, the url path should be **`/#contact-me`**. Also, the click should scroll to the corresponding section with a smooth animation. The code for that has been provided for you via the **`handleClick`** function. You need to hook that function with the **`a tag`** **`onClick`** event. Bear in mind the Projects section has an **`id`** called **`projects-section`** and the Contact Me section has an **`id`** called **`contactme-section`**.

Avoid any key related warnings when opening the console.

The final header UI should look like below:

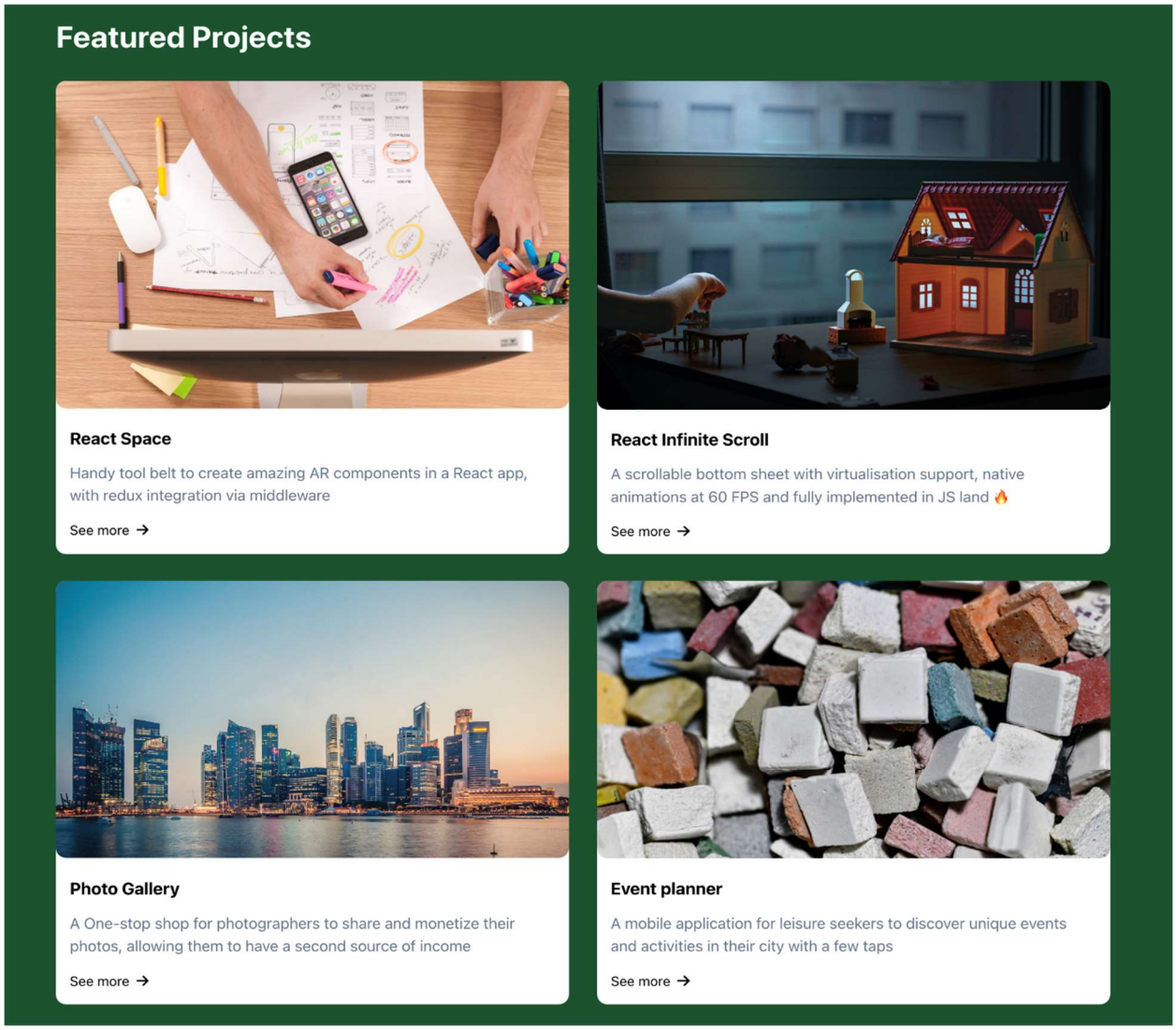
Step 2

Open the **`LandingSection.js`** file. Implement the below UI to provide a landing section for the app with an avatar, a greeting and a brief role description. For the data, use the variables provided at the top of the file (greeting, bio1 and bio2) and not personal data. For the avatar you can use the next url: **`https://i.pravatar.cc/150?img=7`** All the components you need have been already imported for you.



Step 3
Open the **`ProjectsSection.js`** component. This component is already implemented, however the Card component that it uses to display information about each project it's not. The Projects Section component already defines a projects array with the data for each project and that information is passed to each Card component as props.

Open the **`Card.js`** component and implement the UI for the card. Each card should look like the image below:



You can use the following components from Chakra UI that have been already imported for you:

- HStack,
- VStack,
- Image,
- Heading,
- Text,

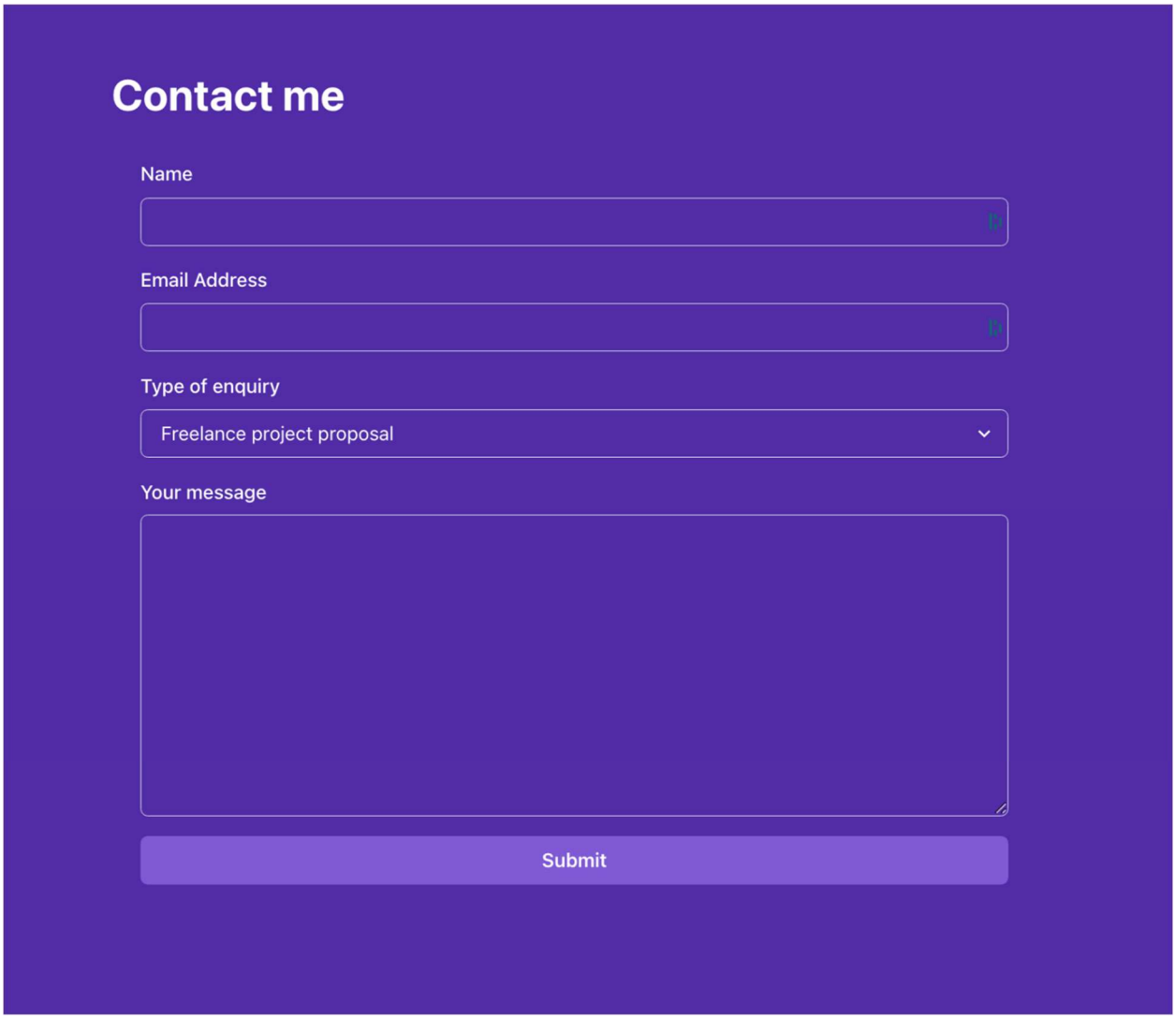
For the right arrow, use the below component. The necessary imports are already provided for you as well. **`<FontAwesomeIcon icon={faArrowRight} size="1x" />`**

Avoid any key related warnings when opening the console.

Step 4
Open the **`ContactMeSection.js`** component. Implement the remaining requirements of the form according to the below specifications.

The form contains 4 input fields: name, email address, type of enquiry and message.

The whole UI of the form is defined for you. You need to implement some missing business logic.



a) **Add the proper configuration to the `useFormik` hook, passing an object with 3 properties**: **`initialValues`**, **`onSubmit`** and **`validationSchema`**.

The `initialValues` object should have the following fields:

- `firstName`
- `email`
- `type`
- `comment`

The `onSubmit` function should perform an API call by using the submit helper from `useSubmit` hook. Inspect the `useSubmit` custom hook to see the arguments the submit function expects.

The `validationSchema` should be a Yup schema that validates the form fields. The validation rules are as follows:

- `firstName`
- `email`
- `type`
- `comment`

b) **Make the Input components from Chakra UI controlled components**.

`useFormik` hook returns an object with a function called `getFieldProps` that when called, returns an object with the necessary props to make the input controlled.

c) **Show the error messages for each field when the field is touched and the validation fails**.

Each field is grouped in a `FormControl` component. The `FormControl` component takes a isInvalid prop that you can use to show the error message.

The `isInvalid` prop should be true when the field is touched and the validation fails.

The `FormErrorMessage` component from Chakra UI should display the corresponding error message if the `isInvalid` prop from the parent `FormControl` component is true.

Below is an example of how the UI should be displayed when the validation fails for the `firstName` field:

d) **Connect the form `onSubmit` prop with Formik's `handleSubmit` function.**

Make sure the default HTML form behaviour is prevented when a submission occurs.

e) **Show an alert when the form is submitted successfully**.

You need to listen to changes in the response object from the **`useSubmit`** hook. Also, when the form is submitted, a loading indicator should be shown in the Submit button. You can use the **`isLoading`** property from the **`useSubmit`** hook.

The **`useSubmit`** hook is implemented in a way that 50% of the times it will return a successful response and 50% of the times it will return an error response.

The response object from the API has 2 properties:

- **`type`**: 'success' | 'error'
- **`message`**: Extra contextual information about the response

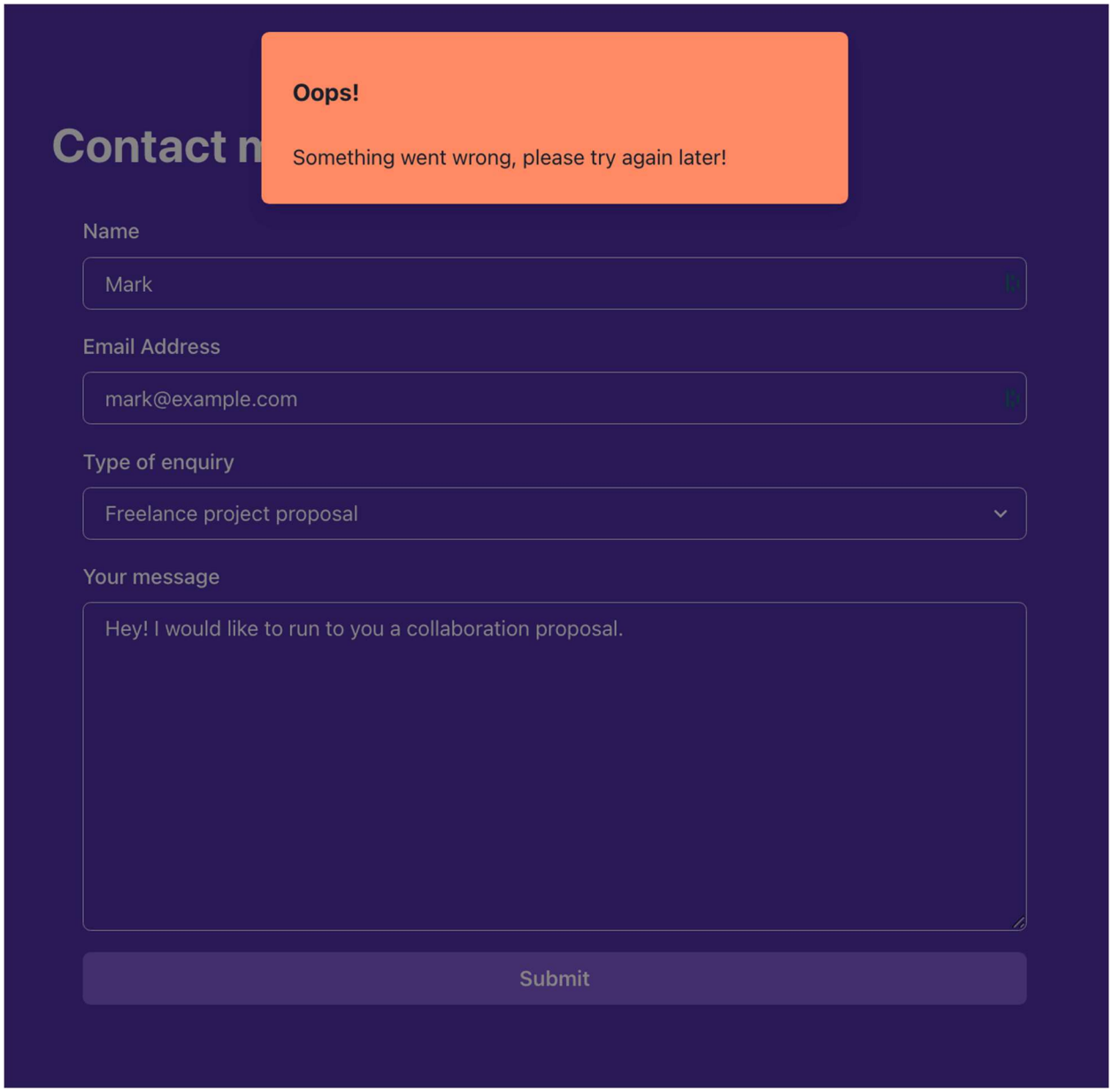
You can use the provided **`useAlertContext`** hook to show the alert. The hook returns a function named **`onOpen`** that you can call to display it.

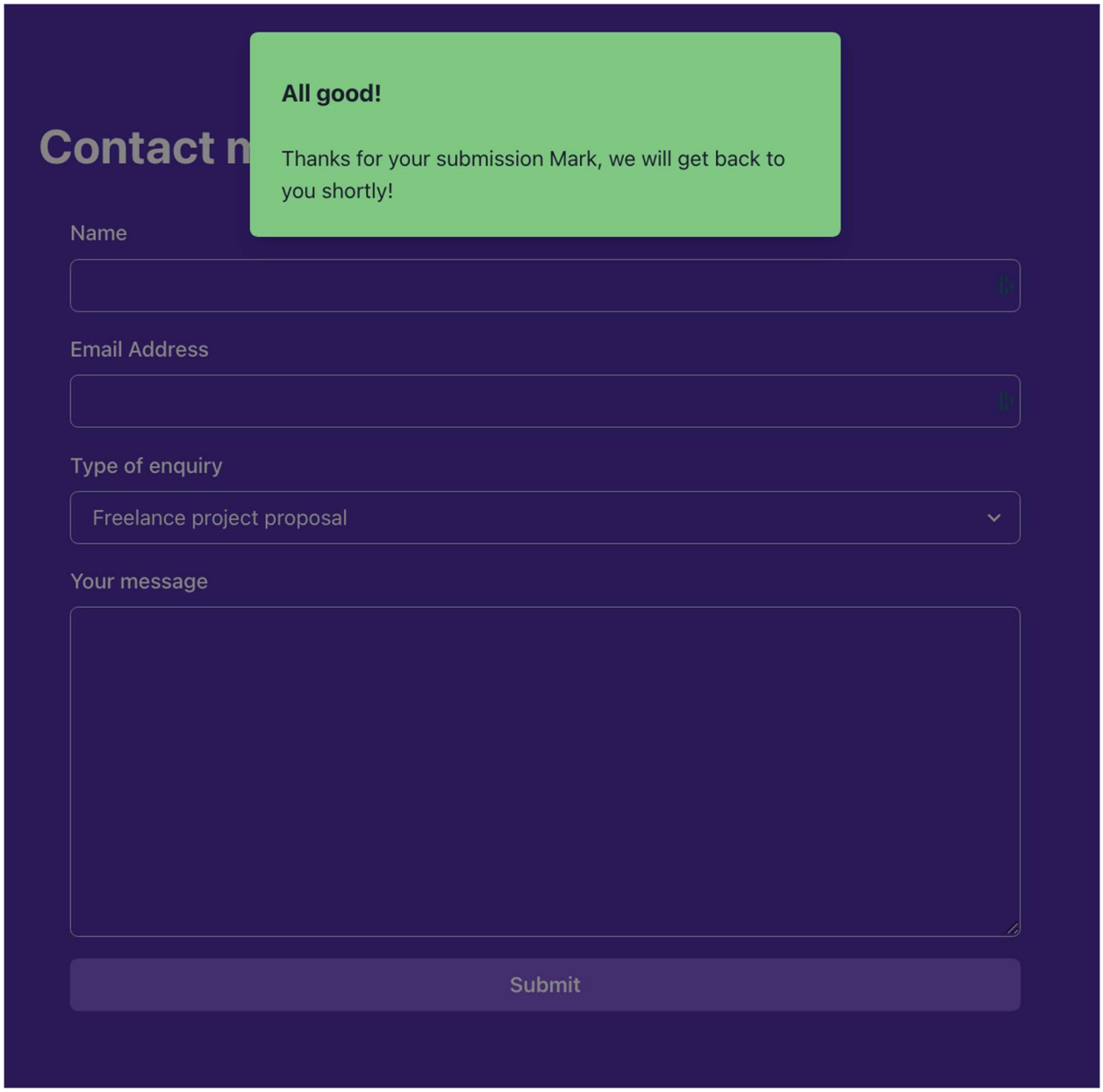
Check the **`alertContext.js`** file to see the arguments the **`onOpen`** function expects.

If the response is successful, the alert should display in its content the first name of the user, according to the value typed in the form first field, so make sure you are passing the right arguments to the submit function returned from the **`useSubmit`** hook.

In addition, the form has to be reset if the response is successful. To achieve this, use the **`resetForm`** function from the object returned from the **`useFormik`** hook.

This is how the UI should look for both cases:


Oops!
Something went wrong, please try again later!
Contact m
Name
Mark
Email Address
mark@example.com
Type of enquiry
Freelance project proposal
Your message
Hey! I would like to run to you a collaboration proposal.
Submit



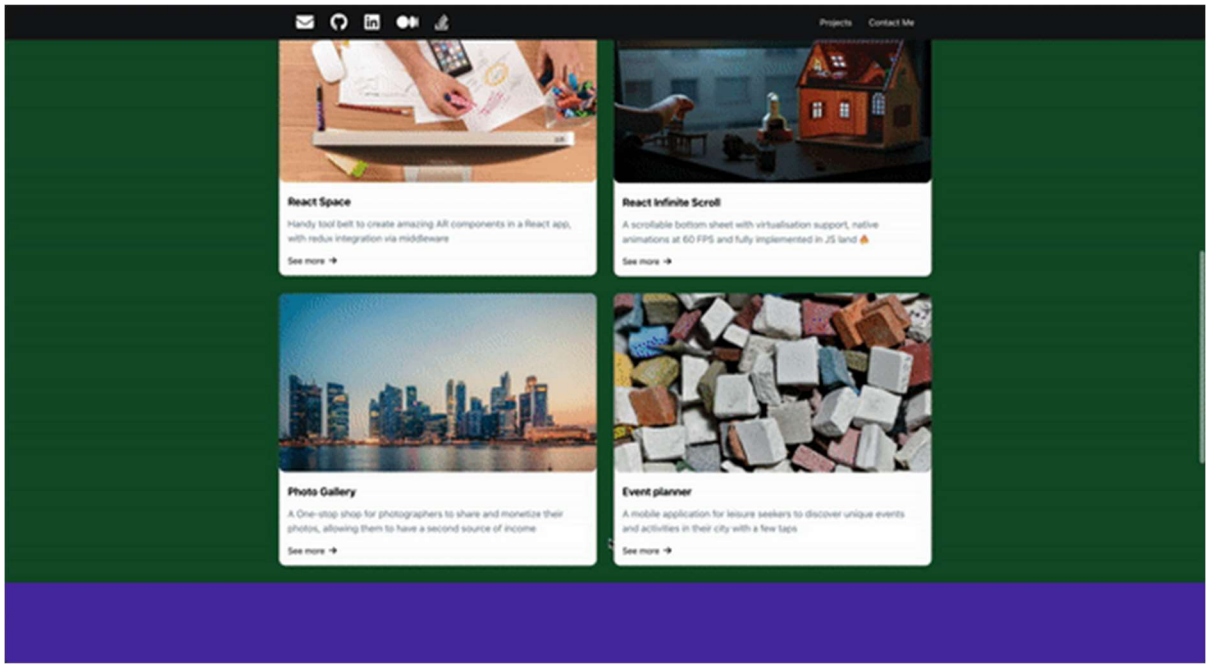
Step 5 (bonus)
Implement a header show-hide animation depending on the scroll direction. The header should slide up with some animation and be hidden when scrolling down the page. When scrolling up, the header should slide down and be visible.

For that, the outermost Box component has some transition properties already defined. Your logic should change the transform property of the underlying Box DOM element depending on the scroll direction.

When scrolling up, the transform style property from the Box DOM element should be translateY(0). When scrolling down, the transform style property from the Box DOM element should be **`translateY(-200px)`**.

Here are some of the elements you may need for the implementation:

- The **`useEffect`** hook
- The **`useRef`** hook
- Setting up listeners for the scroll **`eventwindow.addEventListener('scroll', handleScroll)`**
- Removing listeners for the scroll **`event:window.removeEventListener('scroll', handleScroll)`**
- Keeping track of the previous scroll position in a variable



Solution code

Here is the completed solution code for the App.js file:

```
import { ChakraProvider } from "@chakra-ui/react";
import Header from "./components/Header";
import LandingSection from "./components/LandingSection";
import ProjectsSection from "./components/ProjectsSection";
import ContactMeSection from "./components/ContactMeSection";
import Footer from "./components/Footer";
import { AlertProvider } from "./context/alertContext";
import Alert from "./components/Alert";

function App() {
 return (
   <ChakraProvider>
     <AlertProvider>
       <main>
         <Header />
         <LandingSection />
         <ProjectsSection />
         <ContactMeSection />
         <Footer />
         <Alert />
       </main>
     </AlertProvider>
   </ChakraProvider>
 );
}

export default App;
```

Here is the completed solution code for the context/alertContext.js file:

```
import {createContext, useContext, useState} from "react";

const AlertContext = createContext(undefined);

export const AlertProvider = ({ children }) => {
  const [state, setState] = useState({
    isOpen: false,
    type: 'success',
```

```
    message: '',
  });

  return (
    <AlertContext.Provider
      value={{
        ...state,
        onOpen: (type, message) => setState({ isOpen: true, type, message }),
        onClose: () => setState({ isOpen: false, type: '', message: '' }),
      }}
    >
      {children}
    </AlertContext.Provider>
  );
};

export const useAlertContext = () => useContext(AlertContext);
```

Here is the completed solution code for the components/Header.js file:

```
import React, { useEffect, useRef } from "react";
import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";
import { faEnvelope } from "@fortawesome/free-solid-svg-icons";
import {
 faGithub,
 faLinkedin,
 faMedium,
 faStackOverflow,
} from "@fortawesome/free-brands-svg-icons";
import { Box, HStack } from "@chakra-ui/react";

const socials = [
 {
   icon: faEnvelope,
   url: "mailto: hello@example.com",
 },
 {
   icon: faGithub,
   url: "https://www.github.com/sureskills",
 },
 {
   icon: faLinkedin,
   url: "https://www.linkedin.com/in/sureskills/",
 },
```

```
 {
   icon: faMedium,
   url: "https://medium.com/@sureskills",
 },
 {
   icon: faStackOverflow,
   url: "https://stackoverflow.com/users/sureskills",
 },
];

/**
* This component illustrates the use of both the useRef hook and useEffect hook.
* The useRef hook is used to create a reference to a DOM element, in order to twe
ak the header styles and run a transition animation.
* The useEffect hook is used to perform a subscription when the component is moun
ted and to unsubscribe when the component is unmounted.
* Additionally, it showcases a neat implementation to smoothly navigate to differ
ent sections of the page when clicking on the header elements.
*/
```

Here is the completed solution code for the components/Card.js file:

```
import { Heading, HStack, Image, Text, VStack } from "@chakra-ui/react";
import { FontAwesomeIcon } from "@fortawesome/react-fontawesome";
import { faArrowRight } from "@fortawesome/free-solid-svg-icons";
import React from "react";

const Card = ({ title, description, imageSrc }) => {
   return (
     <VStack
       color="black"
       backgroundColor="white"
       cursor="pointer"
       borderRadius="xl"
     >
       <Image borderRadius="xl" src={imageSrc} alt={title} />
       <VStack spacing={4} p={4} alignItems="flex-start">
         <HStack justifyContent="space-between" alignItems="center">
           <Heading as="h3" size="md">
             {title}
           </Heading>
         </HStack>
         <Text color="#64748b" fontSize="lg">
           {description}
         </Text>
```

```
          <HStack spacing={2} alignItems="center">
            <p>See more</p>
            <FontAwesomeIcon icon={faArrowRight} size="1x" />
          </HStack>
        </VStack>
      </VStack>
    );
};

export default Card;
```

Here is the completed solution code for the components/Alert.js file:

```
import {
 AlertDialog,
 AlertDialogBody,
 AlertDialogContent,
 AlertDialogHeader,
 AlertDialogOverlay,
} from "@chakra-ui/react";
import { useAlertContext } from "../context/alertContext";
import { useRef } from "react";

/**
* This is a global component that uses context to display a global alert message.

*/
function Alert() {
 const { isOpen, type, message, onClose } = useAlertContext();
 const cancelRef = useRef();
 const isSuccess = type === "success"

 return (
   <AlertDialog
     isOpen={isOpen}
     leastDestructiveRef={cancelRef}
     onClose={onClose}
   >
     <AlertDialogOverlay>
       <AlertDialogContent py={4} backgroundColor={isSuccess ? '#81C784' : '#FF8A
65'}>
         <AlertDialogHeader fontSize="lg" fontWeight="bold">
           {isSuccess ? 'All good!' : 'Oops!'}
         </AlertDialogHeader>
```

```
          <AlertDialogBody>{message}</AlertDialogBody>
        </AlertDialogContent>
      </AlertDialogOverlay>
    </AlertDialog>
  );
}

export default Alert;
```

Here is the completed solution code for the components/Footer.js file:

```
import React from "react";
import {Box, Flex} from "@chakra-ui/react";

const Footer = () => {
  return (
    <Box backgroundColor="#18181b">
      <footer>
        <Flex
          margin="0 auto"
          px={12}
          color="white"
          justifyContent="center"
          alignItems="center"
          maxWidth="1024px"
          height={16}
        >
          <p>Pete • © 2022</p>
        </Flex>
      </footer>
    </Box>
  );
};

export default Footer;
```

Here is the completed solution code for the components/FullScreenSection.js file:

```
import * as React from "react";
import { VStack } from "@chakra-ui/react";

/**
* Illustrates the use of children prop and spread operator
*/
const FullScreenSection = ({ children, isDarkBackground, ...boxProps }) => {
 return (
   <VStack
     backgroundColor={boxProps.backgroundColor}
     color={isDarkBackground ? "white" : "black"}
   >
     <VStack maxWidth="1280px" minHeight="100vh" {...boxProps}>
       {children}
     </VStack>
   </VStack>
 );
};

export default FullScreenSection;
```

Here is the completed solution code for the components/LandingSection.js file:

```
import React from "react";
import { Avatar, Heading, VStack } from "@chakra-ui/react";
import FullScreenSection from "./FullScreenSection";

const greeting = "Hello, I am Pete!";
const bio1 = "A frontend developer";
const bio2 = "specialized in React";

const LandingSection = () => (
 <FullScreenSection
   justifyContent="center"
   alignItems="center"
   isDarkBackground
   backgroundColor="#2A4365"
 >
   <VStack spacing={16}>
     <VStack spacing={4} alignItems="center">
       <Avatar
         src="https://i.pravatar.cc/150?img=7"
```

```
          size="2xl"
          name="Your Name"
        />
        <Heading as="h4" size="md" noOfLines={1}>
          {greeting}
        </Heading>
      </VStack>
      <VStack spacing={6}>
        <Heading as="h1" size="3xl" noOfLines={1}>
          {bio1}
        </Heading>
        <Heading as="h1" size="3xl" noOfLines={1}>
          {bio2}
        </Heading>
      </VStack>
    </VStack>
  </FullScreenSection>
);

export default LandingSection;
```

Here is the completed solution code for the components/ProjectsSection.js file:

```
import React from "react";
import FullScreenSection from "./FullScreenSection";
import { Box, Heading } from "@chakra-ui/react";
import Card from "./Card";

const projects = [
  {
    title: "React Space",
    description:
      "Handy tool belt to create amazing AR components in a React app, with redux
integration via middleware",
    getImageSrc: () => require("../images/photo1.jpg"),
  },
  {
    title: "React Infinite Scroll",
    description:
      "A scrollable bottom sheet with virtualisation support, native animations at
 60 FPS and fully implemented in JS land 🔥",
    getImageSrc: () => require("../images/photo2.jpg"),
  },
  {
```

```
   title: "Photo Gallery",
   description:
     "A One-
stop shop for photographers to share and monetize their photos, allowing them to
have a second source of income",
   getImageSrc: () => require("../images/photo3.jpg"),
 },
 {
   title: "Event planner",
   description:
     "A mobile application for leisure seekers to discover unique events and acti
vities in their city with a few taps",
   getImageSrc: () => require("../images/photo4.jpg"),
 },
];

const ProjectsSection = () => {
 return (
   <FullScreenSection
     backgroundColor="#14532d"
     isDarkBackground
     p={8}
     alignItems="flex-start"
     spacing={8}
```

Here is the completed solution code for the components/ContactMeSection.js file:

```
import React, {useEffect} from "react";
import { useFormik } from "formik";
import {
 Box,
 Button,
 FormControl,
 FormErrorMessage,
 FormLabel,
 Heading,
 Input,
 Select,
 Textarea,
 VStack,
} from "@chakra-ui/react";
import * as Yup from 'yup';
import FullScreenSection from "./FullScreenSection";
import useSubmit from "../hooks/useSubmit";
```

```
import {useAlertContext} from "../context/alertContext";

/**
* Covers a complete form implementation using formik and yup for validation
*/
const ContactMeSection = () => {
 const {isLoading, response, submit} = useSubmit();
 const { onOpen } = useAlertContext();

 const formik = useFormik({
   initialValues: {
     firstName: "",
     email: "",
     type: "hireMe",
     comment: "",
   },
   onSubmit: (values) => {
     submit('https://john.com/contactme', values);
   },
   validationSchema: Yup.object({
     firstName: Yup.string().required("Required"),
     email: Yup.string().email("Invalid email address").required("Required"),
     comment: Yup.string()
```

Here is the completed solution code for the hooks/useSubmit.js file:

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

const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

/**
* This is a custom hook that can be used to submit a form and simulate an API cal
l
* It uses Math.random() to simulate a random success or failure, with 50% chance
of each
*/
const useSubmit = () => {
 const [isLoading, setLoading] = useState(false);
 const [response, setResponse] = useState(null);

 const submit = async (url, data) => {
   const random = Math.random();
   setLoading(true);
   try {
```

```
      await wait(2000);
      if (random < 0.5) {
        throw new Error("Something went wrong");
      }
      setResponse({
        type: 'success',
        message: `Thanks for your submission ${data.firstName}, we will get back t
o you shortly!`,
      })
    } catch (error) {
      setResponse({
        type: 'error',
        message: 'Something went wrong, please try again later!',
      })
    } finally {
      setLoading(false);
    }
  };

  return { isLoading, response, submit };
}

export default useSubmit;
```

In a previous video, you were introduced to a possible solution for the portfolio page, where most of the concepts you learned over the duration of this course were applied in one way or another. However, there are still some interesting extras about the solution that will be illustrated in this reading.

Header animation

In the Header.js component, there are two React core hooks being used: **`useRef`** and **`useEffect`**.

Those two are used in conjunction to achieve the smooth animation of the header. If you run the application, you can see that the header hides when I am scrolling down, and shows up when I am scrolling back up.

To implement this behavior, I have to use a side effect and subscribe to the scroll event on the window object using **`window.addEventListener`**.

It's important to remove all subscriptions before the unmounting phase. For that, I have to return a function inside useEffect that performs that task. That's the **`window.removeEventListener`** call you see executed inside that function.

```
useEffect(() => {
  const handleScroll = () => {
    // Business logic
  };

  window.addEventListener('scroll', handleScroll);

  return () => {
    window.removeEventListener('scroll', handleScroll);
  }
}, []);
```

To animate the header, you need to deal with its underlying DOM node and apply some style transition. Do you recall the React way to do that? If you said **`useRef`**, you guessed right! That's what I am doing on the container **`Box`** and **`headerRef`** holds a reference to the underlying **`<div>`** node.

```
const Header = () => {
  const headerRef = useRef(null);

  …
  return (
    <Box
      ref={headerRef}
      {...}
    >
      …
    </Box>
  );
};
```

Finally, **`handleScroll`** is the handler function that will be called every time there is a change in the vertical scroll position.

The meat of this function resides in the comparison between the previous value and the new value. That determines the direction of the scroll and which style I should apply in order to either show or hide the header. Since I am using transition properties in the container **`Box`** component, the change is animated.

```
useEffect(() => {
  let prevScrollPos = window.scrollY;

  const handleScroll = () => {
    const currentScrollPos = window.scrollY;
    const headerElement = headerRef.current;
    if (!headerElement) {
      return;
    }
    if (prevScrollPos > currentScrollPos) {
      headerElement.style.transform = "translateY(0)";
    } else {
      headerElement.style.transform = "translateY(-200px)";
    }
    prevScrollPos = currentScrollPos;
  }

  window.addEventListener('scroll', handleScroll)

  return () => {
    window.removeEventListener('scroll', handleScroll)
  }
}, []);


…
  return (
    <Box
      position="fixed"
      top={0}
      left={0}
      right={0}
      translateY={0}
      transitionProperty="transform"
      transitionDuration=".3s"
      transitionTimingFunction="ease-in-out"
      backgroundColor="#18181b"
      ref={headerRef}
    >
     …
    </Box>
```

Header navigation

There is another neat trick I would like to show you, which also happens in the Header component.

Let's see what happens when I click on one of the header sections. Do you see how it nicely animates and scrolls into its position on the page? Let me show you how simple it is to implement something like that. Coming back to the code, I have this **`handleClick`** function that is invoked when I click on one of the header navigation items, either Projects or Contact Me.

```
const handleClick = (anchor) => () => {
  const id = `${anchor}-section`;
  const element = document.getElementById(id);
  if (element) {
    element.scrollIntoView({
      behavior: "smooth",
      block: "start",
    });
  }
};
```

I have defined some **`id`**s in other sections of the page. For instance, the header of the projects section has an **`id`** called project-section. The **`handleClick`** function is called with the anchor name depending on where the navigation should happen, as per the code below:

```
<HStack spacing={8}>
  <a href="#projects" onClick={handleClick("projects")}>
    Projects
  </a>
  <a href="#contactme" onClick={handleClick("contactme")}>
    Contact Me
  </a>
</HStack>
```

To access that DOM element, you can then use **`document.getElementById`** and pass the corresponding ID. Once you have it, you can call **`element.scrollIntoView`** with an object as parameter, setting behavior as smooth and block start. Nice and simple, isn't it?

Formik and Yup validation

Formik works very nicely with Yup, an open source library that allows you to define validation rules in a declarative way. Let's break down in detail the rules set for the Contact Me form, as part of the **`useFormik`** hook. **`useFormik`** hook comes with a **`validationSchema`** option as part of its configuration object.

```
const formik = useFormik({
  initialValues: {
    firstName: "",
    email: "",
    type: "hireMe",
    comment: "",
  },
  onSubmit: (values) => {
    submit('https://john.com/contactme', values);
  },
  validationSchema: Yup.object({
    firstName: Yup.string().required("Required"),
    email: Yup.string().email("Invalid email address").required("Required"),
    comment: Yup.string()
      .min(25, "Must be at least 25 characters")
      .required("Required"),
  }),
});
```

For the **`firstName`** field, the rule states that it has to be a string and it can't be empty. If empty, Formik will register an error message with the label "Required".

```
firstName: Yup.string().required("Required"),
```

The email input is also required. Observe how Yup already provides us with common validators out of the box, like one to verify that what users type is a valid email. If incorrect, Formik will register an error on that input with the error message "Invalid email address". Quite straightforward right?

```
email: Yup.string().email("Invalid email address").required("Required"),
```

Finally, I am making the comment field mandatory, with a minimum length of 25 characters.

```
comment: Yup.string()
 .min(25, "Must be at least 25 characters")
 .required("Required"),
```