**React Countries**

**Learning Objectives**

* Be able to make HTTP requests to fetch data inside a React component
* Be able to define stateless functional components
* Further practice passing around data between React components

**Intro**

You all wrote awesome applications using the countries RESTful API in vanilla JS. In this lesson we are going to use React to make a HTTP request to the same API. As we saw with the Comments app, by setting the received data on our state, we trigger a re-render of our application, which we can use to populate our UI. Here we will see how React’s component lifecycle methods can help us perform HTTP requests. We will also be using stateless functional components where possible.

**Design**

We are going to build an app that makes a request to the countries API, populates a drop-down select with the names of the countries, and allows the user to select a country and see more details about the selected country. Let’s think about which components we might need to make this, and what state and props each would require.

Discuss design and get to a structure that looks something this.

* CountryContainer: state: countries, selectedCountryAlpha3Code
* CountrySelect: props - countries, handleChange
* CountryDetail: props - country

**Implementation**

Let’s start by creating a React project with Create React App:

create-react-app countries

Instructor note: Hand out the src folder from countries\_api\_start and get students to drop it into their own React app. Then ask the class…

1. What containers/components do we already have in the start code?
2. What props and state does each they have?

Answers:

Let’s look at our CountryContainer, which will be our main parent component. This should control the state of our application. Let’s set up the initial state of the Container so that it has an empty list of countries and a selected country alpha3Code which will start as an empty String.

*// CountryContainer.js*

**constructor**(props){

**super**(props);

**this**.state **=** { *// CHANGED*

countries: [],

selectedCountryAlpha3Code: ''

};

}

We have rendered this in our App top-level component.

Go to React Dev tools in the console to see the state of the components.

**Getting Countries from API**

We’re going to use one of the lifecycle methods to perform our HTTP request to the API, componentDidMount. This method will be triggered when the component has successfully been rendered into the DOM. The React documentation recommends that this is the right place to do HTTP requests.

*//CountryContainer.js*

componentDidMount() {

**const** url **=** 'https://restcountries.eu/rest/v2/all';

fetch(url)

.then(res **=>** res.json())

.then(countries **=>** **this**.setState({ countries: countries }))

.**catch**(err **=>** console.error);

}

Task (2 minutes): Use React Dev tools to check that countries andselectedCountryAlpha3Code have been added to the state of CountriesContainer

Again we can check dev tools and see that the state has changed!

**Creating a select**

Now we can start adding to our other components. Let’s have a look at our countries select dropdown - currently there is no information in it.

Inspecting the CountrySelector component with the React Dev tools, we can see that it doesn’t have any props. What properties would we like this to have? The list of countries we just stored in our state. Let’s set that as a prop on it.

*// CountryContainer.js*

render(){

**return**(

**<**div**>**

**<**h2**>**Country Container**<**/h2>

**<**CountrySelector

countries**=**{**this**.state.countries} />

**<**CountryDetail **/>**

**<**/div>

);

}

The CountrySelector component now has access to the array of countries, so let’s set it up to use that data to populate the select with option elements displaying the countries’ names.

Ask the class: How might we populate the select with an option for each country?

Answer:

*//CountrySelector.js*

**const** CountrySelector **=** (props) **=>** {

**const** options **=** props.countries.map(country **=>** {

**return** **<**option value**=**{country.alpha3Code} key**=**{country.alpha3Code}**>**{country.name}**<**/option>

})

**return** (

**<**select name**=**"country-selector" id**=**"country-selector"**>**

**<**option disabled selected**>**Choose a country...**<**/option>

{ options }

**<**/select>

)

};

**Handling Select onChange()**

We’ll write a function within our stateless component to handle a change in the <select> box. For now, all it will do is log the value. Then we add an onChange attribute to the select:

*// CountrySelector.js*

**const** CountrySelector **=** (props) **=>** {

*// ...*

**function** handleChange(event) {

console.log(event.target.value);

}

**return** (

**<**select name**=**"country-selector" id**=**"country-selector" onChange**=**{handleChange}**>**

*// ...*

)

};

**Passing the callback via props**

Now we have access to the index of the selected country in the countries array. But we only have access to this index in the CountriesSelect component. We want to have access to it at the top level, in the CountryContainer, so that we can use it to set the state of the current country, and cause a re-render of the CountryDetail. Let’s do this with a function that is defined in the CountryContainer and passed as a prop to the CountrySelector. The function is going to take in the index and set it on the CountryContainer’s state.

*// CountryContainer.js*

handleCountrySelected(alpha3Code) {

**this**.setState({ selectedCountryAlpha3Code: alpha3Code })

}

REMEMBER to bind it in the CountryContainer constructor method:

*// CountryContainer.js*

**constructor**(props){

*// ...*

**this**.handleCountrySelected **=** **this**.handleCountrySelected.bind(**this**);

}

…and pass that function to CountrySelector as a prop:

render(){

**return** (

{*/\* ... \*/*}

**<**CountrySelector

countries**=**{**this**.state.countries}

onCountrySelected**=**{**this**.handleCountrySelected}/>

{*/\* ... \*/*}

);

}

Back in our CountrySelector component, we can pass the value of the select into this function as the alpha3Code:

*// CountrySelector.js*

**function** handleChange(event) {

props.onCountrySelected(event.target.value);

}

We can now use the React dev tools to check that the state in the CountryContainer is updating with the selected country alpha3Code.

**CountryDetail Display**

Now that we have a selector that is updating the selected country, the final piece in the puzzle is our detailed display.

When the user selects a country, we want the details of the selected country to be displayed. The CountryDetail component is going to be responsible for this.

So how are we going to do render these details? Now that we have a selector that is updating the selectedCountryAlpha3Code in CountryContainer’s state, we can find the selected country and pass this down to CountryDetail as a prop.

*// CountryContainer.js*

render(){

**const** selectedCountry **=** **this**.state.countries.find(country **=>** country.alpha3Code **===** **this**.state.selectedCountryAlpha3Code);

**return** (

{*/\* ... \*/*}

**<**CountryDetail country**=**{ selectedCountry } />

{*/\* ... \*/*}

);

}

Now we have access to the currently selected country in the CountryDetail component, we can get it to render its various properties.

*// CountryDetail.js*

**const** CountryDetail **=** (props) **=>** {

**return** (

**<**h3**>**{props.country.name}**<**/h3>

);

}

The first time the CountryDetail component is rendered, the HTTP request won’t have been made yet and no country will have been selected by the user, so props.country will be undefined. If we ask an undefined object for the property name, we will get an error in the browser. To avoid this, we will put in a guard to return out of the function and not render anything if props.country is falsy.

*// CountryDetail.js*

**const** CountryDetail **=** (props) **=>** {

**if** (**!**props.country) **return** **null**; *// UPDATED*

**return** (

**<**h3**>**{props.country.name}**<**/h3>

);

}

**Task: (5 minutes)**

Render some more of the country’s properties in the CountryDisplay component.

Example solution

**Recap**

Task (10 minutes): Write down the order in which the render and componentDidMount methods are being called as the components are created and the user interacts with them. Take note of the state and props of each component at each stage.

Answer:

**Conclusion**

We can now build a multi component React app, and hook into a React component’s lifecycle methods to make an HTTP request and update its own state, triggering re-rendering of the UI on the completion of the request using the setState method.