# Introduction to React

### Learning Objectives

* Understand the benefits of using a framework
* Know what React is
* Explain some of the key features of React
* Be able to create an application using Create React App

## Introduction

### Frameworks

Both back-end and front-end frameworks provide our applications with structure and often handle common tasks for us. Frameworks are designed to make building an application quicker and easier, and help us to achieve a result which is easier to reason about and therefore easier to maintain. However, in order to use any framework, we have to learn how to learn how it wants us to do things.

Each framework has its own way of doing things; some are more opinionated than others but once we’ve learned how to talk in a framework’s language, we can leverage its functionality, focus on the core of our application and the logic we need to write.

What is the difference between a framework and a library?

* A framework is a type of library; it is an organisational library, and will often contain methods that we can use, but don’t always call ourselves.
* A library is a set of methods and objects that we can use to help us complete programming tasks.

### JavaScript Front-end Frameworks

Writing well-structured vanilla JavaScript applications, with a clear separation of concerns, it difficult to achieve. There are also common tasks that we want to carry out, such as manipulating the DOM, which become repetitive and time consuming. For these reasons, developers over the years have created libraries and frameworks that help with web development. We are going to be using the front-end framework React to help us with our front-end development.

## React

### Popularity

React says that it is ‘A JavaScript library for building user interfaces’. It is a front-end framework, created and used by Facebook. Making applications that are fast and easier to develop has made React a very popular choice in the web community and it is widely used as shown by the following:

* [A list of companies that use React](https://stackshare.io/)
* [State of JavaScript 2019 - Front-end](https://2019.stateofjs.com/front-end-frameworks/)

### Key Features

If we go to the [React the website](https://facebook.github.io/react/), we can see there are three main characteristics of React:

* One Way Data Flow
* Virtual DOM
* Component based UI - ‘HTML’ in JS with JSX

This week we are going to be using React to help us build well-structured view logic, using its DOM manipulation methods to make reading and writing to the DOM more efficient than we have previously been able to achieve with vanilla JavaScript.

#### One way flow

React takes a one-way flow approach. All of the data your app uses - its state - is kept in one place and then passed down through a hierarchy/tree of components for them to use as needed.

When a change happens, we alter the total state of our app and then just do the whole initial display again. We don’t need to worry about how each little change alters our UI, we just render the whole thing again, passing down the new data.

You might be thinking this incredibly inefficient. And it would be if it wasn’t for the middle item on that list, the virtual DOM.

#### Virtual DOM

Updating the DOM using JS is an expensive operation, it tasks a huge amount of time compared to us multiplying numbers, or searching a reasonably sized array. If after every little change our React application re-rendered the whole DOM, our app would be very inefficient and very slow.

React applications are known for being fast. So, how does it achieve this? React keeps a virtual version of the DOM in memory. When we make a change it updates this and compares it to the real DOM (a diff). It only updates the real DOM where necessary.

It is this step which allow us to write fast efficient applications by only having to describe a simple render.

#### Component based UI

React encourages us to break down our user interface into smaller parts called components. (By doing this we can make sure each part is only looking after one thing… which software principle does this relate to? Single Responsibility principle!) These can be reusable and are really helpful in separating the functionality of your app into independent parts.

These components may (but don’t have to) use a React-specific syntax called JSX. JSX allows us to use HTML-like syntax when describing new elements, whether HTML elements or React components, which saves us time and can make our React code more readable. Some frameworks use a templating language for a similar purpose, writing HTML which can pull in information from a separate JavaScript file, but React allows us to do all of this in one place. (Some people prefer to have those things separate, it depends on the nature of your app as to what’s most appropriate.)

## Create React App

When creating a React application we can configure our application manually, setting up our own server, installing React and any other packages we might need to get it up and running. However, Facebook have produced a package called ‘Create React App’ that allow us to create a boilerplate application with all the build configuration already complete. It comes with a number of useful features already configured:

* A built-in web server (so we don’t need to set up Express)
* A build script that includes a pre-configured version of Webpack
* Babel (Transpiles our code at the point of bundling, so we can use all the latest language features without worrying if they are supported by our users’ browsers.)
* The ability to import CSS and images
* Hot reloading (Injects changes to your source code into the current state of the page, so we don’t need to refresh the browser each time you make a change.)
* Easily deployable to many hosting providers, for example Github Pages
* [Jest](https://facebook.github.io/jest/) - a testing framework that lets us test our app in various ways

…and lots more.

### Creating an Application with Create React App

Let’s make use of these great features and create a “Hello World!” application with Create React App.

To use create-react-app to give us a start-point, we can use npx to get a temporary version of create-react-app to use to create the application.

npx create-react-app hello\_world

cd hello\_world

npm start

Note: An alternative to using the npx prefix would be to globally install create-react-app and then use npm create-react-app hello\_world

The npm start command has bundled the source code and started the server, so we don’t need to run webpack separately anymore. It should have also opened the application in a new browser tab going to http://localhost:3000.

### Create-React-App Application Architecture

Let’s take a look at the application architecture. Some of the files and directories should look familiar. There is the ‘package.json’, ‘.gitignore’ and ‘node\_modules’ in the top level directory and there are also the ‘public’ and ‘src’ directories. The ‘src’ directory is where our source code lives, and this is where we will be putting our React components.

#### index.js: The application’s entry point

The entry point to the application is index.js, which renders the top level React component to the page. The ReactDOM’s render method takes the React component to be rendered, and the DOM element it should be appended to. The result of the render method is very similar to when we appended elements to the DOM by using element.appendChild.

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

index.js accesses the element with an id of ‘root’ using getElementById from the DOM. (getElementById is just another method for querySelector). The ‘root’ element is defined in the public/index.html.

In most cases, the is the only time we will be directly accessing an element from the DOM in our React applications. Instead, we use a component rendering hierarchy, rendering the top level component to the page and then letting React handle the rest of the DOM manipulation for us. Therefore we will rarely (if ever) be using the document method createElement, or the element methods appendChild or textContent.

## Components

So we know that the component defined in App.js is being rendered to the page by index.js. Let’s have a look at it. It is created as a function. This is actually a special kind of React component called a Stateless Functional Component which we will cover later.

Every React Component must implement a render method that returns a DOM element, an array of DOM elements or null. (We will be looking more at this new syntax in a moment.) As our App component is an SFC the function itself takes the place of the render method.

### Deleting Unneeded Boilerplate Code

Create React App comes with some boilerplate code that we don’t need (you can see some of it being used to render information on the page). Let’s start by removing some of the unnecessary code and files. We don’t need the React logo file so we can start by deleting that.

rm src/logo.svg

Then is App.js we can delete the import of the logo file.

*// src/App.js*

**import** React **from** 'react';

*// DELETED*

**import** './App.css';

If we look at what the component returns from its function, we can see there is some new syntax there. It looks like HTML, but we know that it’s not because it is inside a JavaScript file. This syntax is called JSX and is what we can use in React to describe what we want to be rendered to the page. We are going to be looking more at JSX next. For now let’s delete the boilerplate code, so we can write our own.

**import** React **from** 'react';

**import** './App.css';

**function** App() {

**return** (

*// DELETED*

);

}

**export** **default** App;

### render

As mentioned before, the Component’s function must return either a DOM element, an array of DOM elements or null. We could use React’s method to create a DOM element, createElement that takes 3 arguments:

* the element tag
* an object containing any classes or ids (in this case null)
* the content

**function** App() {

**return** (

React.createElement('h1', **null**, 'Hello World!')

);

}

If you check the browser, you should now see ‘Hello World!’ instead of the React logo.

## JSX

React provides us with an alternative syntax, preventing us from having to create elements in this way. The syntax is called JSX. It allows us to describe elements in a way that is very similar to HTML. JSX allows us to construct our user interfaces in a much quicker and more natural fashion, declaring our JSX rather than using JavaScript to manually create, manipulate and append DOM elements.

Let’s refactor our ‘h1’ using JSX. JSX looks a lot like HTML, but there are some key differences. For example, giving an element a class attribute, we use className="my\_class", instead of using class="my\_class" as we do in HTML. (This is because class is a reserved keyword in JavaScript.)

**function** App() {

**return** (

**<**h1 className**=**"title"**>**Hello World**!<**/h1>

);

}

Note: React and JSX are two independent technologies, but JSX was built with React in mind. Create React App has Babel already configured and Babel is transpiling the JSX into JavaScript, making it compatible with browsers.

### JSX Syntax Highlighting

The JSX syntax highlighting might be broken as it’s being interpreted as JavaScript. To fix this you will need to get the Babel package for your editor. For Atom you can use Language Babel: <https://atom.io/packages/language-babel>

## Conclusion

We have seen how to use Create React App to create a start point for our application and render to the screen using “Hello World”.

## Additional Resources

There is also a React implementation for native mobile applications: <https://facebook.github.io/react-native/>

# Piggy Bank Application

**Duration: 90 minutes**

### Learning Objectives

* Be able to create a single component application
* Be able to use the JSX syntax
* Use props and state to managing data flow in the application
* Install and be able to use React Dev Tools

## Introduction

In this lesson we are going to build a Piggy Bank application that displays a total and allows a user to deposit and withdraw an amount, which will update the total. We want the the total to change in response to a user’s interaction and we will use the React framework to do what it does so well - manipulate the DOM easily and efficiently.

Previously we have been dividing up our view logic into separate view files to keep our applications modular, and working with React is no different. Throughout the week you will be separating your UI into separate view files, which in React are called Components. A Component is part of the React framework, and its role is to display a section of our user interface. Well-designed React applications have many small components with a single responsibility. However, in this lesson we will be focussing on learning the syntax used by React, so we will only have one component that describe the UI.

The component in our application will be responsible for displaying and keeping track of the total, as well as displaying a ‘deposit’ and ‘withdraw’ button that will update the state of the component with clicked, updating the displayed total. The React Components has some methods that will help us do this.

## Create Piggy Bank Application

Let’s begin by using create-react-app to give us a start-point.

npx create-react-app piggy\_bank

cd piggy\_bank

npm start

And we will delete the boilerplate code we don’t need:

1. the logo file
2. the logo input in App.js
3. the JSX from App.js’ return method

rm src/logo.svg

*// src/App.js*

**import** React **from** 'react';

*// DELETED*

**import** './App.css';

**function** App() {

**return** (

*// ...*

);

}

**export** **default** App;

If we look at what the component returns from the render function, we can see there is some new syntax there. It looks like HTML, but we know that it’s not because it is inside a JavaScript file. This syntax is called JSX and is what we can use in React to describe what we want to be rendered to the page. We are going to be looking more at JSX next. For now let’s delete the boilerplate code, so we can write our own.

**import** React, { Component } **from** 'react';

**import** './App.css';

**function** App() {

**return** (

*// DELETED*

);

}

**export** **default** App;

And let’s render “Hello World!” to the page to ensure we don’t have any errors at this stage.

**class** App **extends** Component {

render() {

**return** (

**<**h1 className**=**"title"**>**Hello World**!<**/h1> /**/** NEW

);

}

}

We have seen how to render to the page using React. Let’s create a PiggyBank component that is going to be responsible for rendering the UI. Our component rendering hierarchy is going to be App renders PiggyBank.

### Creating a Component

When we create a React Component we use a few es6 language features:

1. Destructuring

When we import from ‘react’, we are importing the whole React object, as well as extracting out a particular part of the package, the Component object. Doing this means we don’t need to type React.Component when we want use it, instead we can just type Component.

1. Class Syntax with Inheritance

We create components using the Class syntax. We will create a PiggyBank class which inherits from React’s Component using the extend keyword, and calling super in the constructor. This gives us access to all of the React Components properties and methods, while rewriting our own custom components.

Note: Under the hood this is still prototypal inheritance, but using the new Class syntax we can inherit from React’s Component to create our own custom Components.

Every React Component must implement a render method that returns a DOM element, an array of DOM elements or null. (We will be looking more at this new syntax inside the render method in a moment.)

touch src/PiggyBank.js

*// src/PiggyBank.js*

**import** React, { Component } **from** 'react';

**class** PiggyBank **extends** Component {

}

**export** **default** PiggyBank;

What is the only method that our Component has to implement? render, which must return a DOM element, an array of DOM elements or null.

**class** PiggyBank **extends** Component {

render() {

**return** (

**<**h1**>**Hello World**!** I am a PiggyBank**<**/h1>

)

}

}

This will not be rendered to the page, until is it rendered by another component. To add it to the rendering hierarchy, we need to tell the App component to render it.

Next, we need to import and use our PiggyBank into our App.js file.

*// src/App.js*

**import** React, { Component } **from** 'react';

**import** PiggyBank **from** './PiggyBank'; *// NEW*

**class** App **extends** Component {

render() {

**return** (

**<**PiggyBank **/>** *// MODIFIED*

);

}

}

**export** **default** App;

Note: all JSX tags must be closed, either with a separate closing tag or self-closed.

We should now see that the PiggyBank is being rendered on the page.

## State and Props

We are now going to look at two mechanisms by which React controls data flow in the application. Components have two type attributes that they can use to store and receive data: state and props.

### Props (or Properties)

Props (or properties) are attributes that are given to a component by their parent component. Props cannot be changed, they are immutable. Props are the way we can pass data from one component to another. Let’s pass a title attribute from the App component to PiggyBank component, so that PiggyBank can display it.

We define the prop in JSX like we set attributes on an HTML component, but we get to define the name of the prop (we will use ‘title’). We can either pass the prop as a string or use the braces ({}) to pass in other JavaScript values.

*// src/App.js*

**import** React, { Component } **from** 'react';

**import** PiggyBank **from** './PiggyBank';

**function** App() {

**return** (

**<**PiggyBank title**=**"Savings Pig" **/>**

);

}

**export** **default** App;

We can now access this ‘title’ prop inside the PiggyBank component from the props object using ‘this.props’. Let’s display it on the page inside the h1. When we are writing JSX, we have to wrap the JavaScript in braces, so that it can be differentiated from the JSX.

*// src/PiggyBank.js*

**class** PiggyBank **extends** React.Component {

render() {

**return** (

**<**div className**=**"bank-box"**>**

**<**h1**>**{**this**.props.title}**<**/h1>

**<**/div>

);

}

}

## State: Displaying a total

State is where a component can store data. State, unlike props, is defined in the component itself and can be changed. When state is changed, React renders the page. Changing (or setting) the state to cause a rerender of the page is the way we can quickly and easily update the page.

We are going to use state to store the Piggy Banks total. The total is a value that will be changed and when it is changed (with a user clicking deposit or withdraw) we want to rerender the page to display the new value.

Let’s set up our initial state, which will be a property of the class called state and will be an object. We want a total property that starts at zero. We can then display this. Our initial state is defined in the constructor of our component class.

*// src/PiggyBank.js*

**class** PiggyBank **extends** Component {

**constructor**(props) { *// NEW*

**super**(props);

**this**.state **=** {

total: 0

}

}

*// ...*

}

We can display this on the page, accessing the total with this.state.total. Because a React Component must either return one DOM element, or an array of DOM elements, we need to wrap our two separate elements in a <Fragment> tag, which we can get from React by using destructuring.

**import** React, { Component, Fragment } **from** 'react'; *// MODIFIED*

**class** PiggyBank **extends** Component {

*// ...*

render() {

**return** (

**<**Fragment**>** *// NEW*

**<**h1**>**{**this**.props.title}**<**/h1>

**<**p**>**Total: £{**this**.state.total}**<**/p> /**/** NEW

**<**/Fragment> /**/** NEW

);

}

}

## Updating State

Now we want to add a button that will increase the total when we deposit money into the piggy bank. Let’s start by adding a ‘Deposit’ button to the render method.

render() {

**return** (

**<**Fragment**>**

**<**h1**>**{**this**.props.title}**<**/h1>

**<**p**>**Total: £{**this**.state.total}**<**/p>

**<**button**>**Deposit**<**/button>

**<**/Fragment>

);

}

When a user clicks on this button, we want an amount to be added to the total, and for this change to be reflected on the page. We can trigger this rerendering of the page by updating the state with React’s method, setState. Every time we call setState render will be called and the page will be updated to reflect the changes.

It might seem inefficient to render every time, but remember React uses the virtual DOM to only update the DOM elements that have changed, keeping it highly performant. There is where we start to see the real power of React. Let’s write the method that is going to be responsible for incrementing the total. It’s going to do this by calling setState.

**class** PiggyBank **extends** Component {

**constructor**(props) {

**super**(props);

**this**.state **=** {

total: 0

};

}

deposit() {

**this**.setState();

}

*// ...*

}

setState takes a callback that will be passed the previous state. We can the increment the value from the previous state. We return a new object, which React will then use to update the state. Because the total already has a ‘total’ property, is will assign the new value to it.

deposit() {

**this**.setState(prevState **=>** {

**return** {total: prevState.total **+** 5};

});

}

Now we call set this on the button’s click event in the JSX using the onClick attribute.

render() {

**return** (

**<**Fragment**>**

**<**h1**>**{**this**.props.title}**<**/h1>

**<**p**>**Total: £{**this**.state.total}**<**/p>

**<**button onClick**=**{ **this**.deposit }**>**Deposit**<**/button>

**<**/Fragment>

);

}

If we now try and click the button on the page, we get an error. This is because we have a context problem. The deposit method is being passed as a callback to the button, therefore inside deposit, this isn’t the PiggyBank class. We want it to be the PiggyBank class, so that it can access the state.

### bind

The way React suggest to solve this problem is to the bind method. bind is a function method (in JavaScript, functions are first class objects and therefore can have methods stored on them). It allows us to call it on a function and pass in whatever context we want it to have. It returns a new function with the specified context. React suggests to create this new version of the function as a property of the class.

*// src/PiggyBank.js*

**constructor**(props) {

...

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

}

Note: There are other solution to the problem, for example binding the function as it’s passed to the onClick with <button onClick={ this.deposit.bind(this) }>Add</button>, or to wrap it in an arrow function with <button onClick={ () => { this.deposit() } }>Add</button> but binding in the constructor is the preferred option (and the one recommended in the React documentation) because the bind only has to happen once, when the class is initially set up. If we put it in our render() method, it would happen each time the JSX re-renders.

Now when we click on the button we should see the page being updated, with the total being incremented by 5 each time.

# Chrome Dev Tools

An other advantage of React is that there are powerful development tools in chrome. Download the following Chrome Dev Tool extension and you will have a new ‘React’ tab in the dev tool. (You might need to reopen the Chrome Dev Tools window after installing it). [Link to React-Dev-Tools] (https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en)

By clicking on the different components in the React Dev Tools we can see any props or state that they have. You can also see it being updated live as the state changes.

### Tasks: (15 minutes)

1. Define the amount to be deposited in the App component and pass it has a prop to the PiggyBank component. PiggyBank’s deposit method should then access the amount off it’s props.
2. Add a ‘withdraw’ button which decreases the total by the amount defined and passed down from the App component, preventing it from going below 0.

### Solutions:

1. Define the amount to be deposited in the App component and pass it has a prop to the PiggyBank component. PiggyBank’s deposit method should then access the amount off it’s props.

*// src/App.js*

**<**PiggyBank

title**=**"Savings Pig"

depositAmount**=**{5} *// NEW*

/>

*// src/PiggyBank.js*

deposit() {

**this**.setState(prevState **=>** {

**return** {total: prevState.total **+** **this**.props.depositAmount}; *// NEW*

});

}

1. Add a ‘withdraw’ button which decreases the total by the amount defined and passed down from the App component, preventing it from going below 0.

*// src/PiggyBank.js*

**constructor**(props) {

*// ...*

**this**.withdraw **=** **this**.withdraw.bind(**this**)

}

withdraw() {

**this**.setState(prevState **=>** {

**let** newAmount **=** prevState.total **-** **this**.props.depositAmount;

**if**(newAmount **<** 0){

newAmount **=** 0;

}

**return** {total: newAmount};

});

}

## Recap

 What method does every React component have to implement? Name two differences between props and state? How do we trigger a rerendering in React?

# Conclusion

React allows us to render our UI as components, providing structure to our front end. It’s use of the virtual DOM and rerendering triggered by the setState() method makes updating the DOM really efficient.

We can now create single component applications, using JSX syntax to render information to the screen. We have seen how to trigger a rerender of the UI by updating a component’s state on user interaction. We have also seen how to pass props down from a parent component.

Lastly, React Dev Tools offers a way of inspecting components’ props and state during development.

## Additional Resources

JSX Gotchas - https://shripadk.github.io/react/docs/jsx-gotchas.html

# React Comments

## Learning Objectives

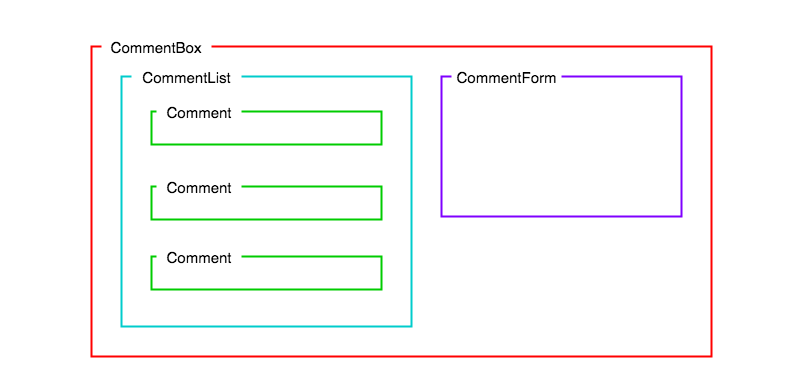
* Create a multiple component React application
* Understand the difference between state and props
* Learn how to pass data between components

## Introduction

We are going to create an application that displays a list of comments and users can add new comments via a form.

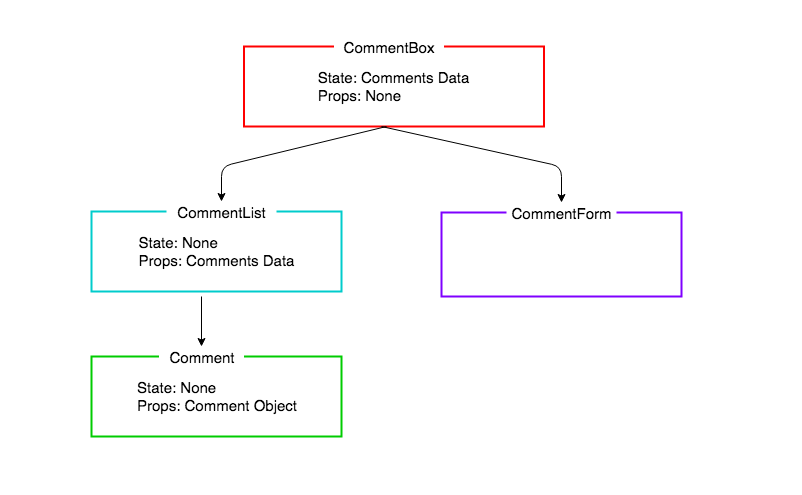
## Application Structure

When planning a React app, we divide up the UI into separate components, with each component responsible for rendering a section of the UI. This is what our component structure is going to look like.



Comments Application Component Structure

CommentsBox will be appended to the HTML root element by React. Then there will be nothing more we need to add to our HTML, the whole application will be drawn by Javascript using React. We do this by describing components for our UI in a component **hierarchy**. Any component rendered by another is called the child of its parent component.



Comments Application Component Hierarchy

In this lesson we will only be concentrating on the left-hand side of the diagram, rendering a list of comments. In the next lesson we will be adding a form, and updating the state, but it’s important to think about the whole application at this point in our planning, because we need to decide which components will have what data for their state and props.

You can see the state and props have already been added to the diagram. The application state is kept high in CommentBox. The reason we keep the state high in our React applications is because of React’s one-way data flow. We do not pass data across the hierarchy chains in React (unlike with other patterns such as PubSub). Therefore, by keeping the comments data at the top level, all the rendering chains can have access to it: the CommentForm can update the data, and the CommentList and Comments can be re-rendered with new comments as they are added to the collection.

As we are not adding the form until the next lesson, in this lesson we will be building a static application that will render based on a hard coded array of comments. This is often a good place to start with a React application.

### Create React App Boilerplate

Let’s use create-react-app to bootstrap our application.

npx create-react-app comments

cd comments

npm start

OK! Through the magic of create-react-app, we’ve pulled down all the bits and pieces we need to start off. We do need to tidy up a little bit, though.

*# cmd-t, new tab*

cd src

rm logo.svg

At this point, the app has broken, because we’ve removed stuff that we’re importing into our app. Let’s fix that. And while we’re at it, let’s remove some default JSX, and render hello world to the page.

**import** React **from** "react";

*// DELETED*

**import** "./App.css";

**function** App() {

**return** (

**<**h1**>**Hello World**!<**/h1> /**/** MODIFIED

);

}

**export** **default** App;

## Containers and Components

We are about to build our first Component, the CommentBox. We can put all our components directly in the ‘src’ folder, but we are going to separate our components into two categories in order to give our application a bit more structure. The two categories are containers and components.

Both will contain React Components, but following convention, if a component contains logic or holds application state, we will put it in the containers directory. If a component just contained presentational code, it will go inside the containers directory.

1. containers - Components with logic or application state
2. components - Components with presentational code

Note: The difference between a container and component is not strictly black and white, but with more experience we will see some more of the differences.

Let’s make those directories:

mkdir src/components

mkdir src/containers

The approach we are going to take it to create the chain of components and check that each one renders the next, before implementing any of the components’ details.

Let’s start with the top of the hierarchy, the CommentBox. Because the CommentBox is going to hold our application state it will go in the containers directory:

touch src/containers/CommentBox.js

*// src/containers/CommentBox.js*

**import** React, { Component } **from** "react";

**class** CommentBox **extends** Component {

render() {

**return** (

**<**div className**=**"comment-box"**>**

Hello, world**!** I am a CommentBox.

**<**/div>

);

}

}

**export** **default** CommentBox;

To get this to render to the page, we need to tell App to render it.

**import** React **from** "react";

**import** CommentBox **from** "./containers/CommentBox"; *// NEW*

**function** App() {

**return** **<**CommentBox **/>** *// MODIFIED*

}

**export** **default** App;

Note: We don’t need the parentheses (()) around the return value, because it is on one line.

Once that is working, we will work our way down the hierarchy, next making the CommentList. The comment list won’t store application state, its sole responsibility will be to be passed the data and render a collection of Comment components.

touch src/components/CommentList.js

*// src/components/CommentList.js*

**import** React, { Component } **from** "react";

**class** CommentList **extends** Component {

render() {

**return** (

**<**div className**=**"comment-list"**>**

Hello, world**!** I am a CommentList.

**<**/div>

);

}

}

**export** **default** CommentList;

For it to be displayed on the page we need to tell our CommentBox component to render it. Let’s also add a title.

*// src/containers/CommentBox.js*

**import** React, { Component } **from** "react";

**import** CommentList **from** "../components/CommentList"; *// NEW*

**class** CommentBox **extends** Component {

render() {

**return** (

**<**div className**=**"comment-box"**>**

**<**h2**>**Comments**<**/h2> /**/** MODIFIED

**<**CommentList **/>** *// NEW*

**<**/div>

);

}

}

**export** **default** CommentBox;

## Comment Component

The last component is the Comment component in the hierarchy and as it is only responsible for receiving data and rendering it to the screen, it will go in the ‘components’ directory.

touch src/components/Comment.js

*// src/components/Comment.js*

**import** React, { Component } **from** "react";

**class** Comment **extends** Component {

render() {

**return** (

**<**div className**=**"comment"**>**

Hello World I am a comment

**<**/div>

);

}

}

**export** **default** Comment;

By the end of this lesson our CommentList is going to dynamically render the collection of comments it’s passed, but for now we are going to hard code a couple of Comment components, to ensure we don’t have any errors in the presentation code.

*// src/components/CommentList.js*

**import** React, { Component } **from** "react";

**import** Comment **from** "./Comment"; *// NEW*

**class** CommentList **extends** Component {

render() {

**return** (

**<**div className**=**"comment-list"**>**

**<**Comment **/>** *// NEW*

**<**Comment **/>** *// NEW*

**<**/div>

);

}

}

**export** **default** CommentList;

We should now see those two comments being rendered to the screen. Next’s let’s populate the comments with come text. Again, by the end of the lesson this content for each comment will be stored in the collection of data, but for now we will hard code the values in, to ensure our Comment component is rendering it properly.

Let’s pass the author and comment to the Comment component as props. We will pass an ‘author’ property as we have been doing with props previously, but the comment text we are going to use a special kind of property, called children. The children props refers to any text or elements passed between the JSX opening and closing tags.

render() {

**return** (

**<**div className**=**"comment-list"**>**

**<**Comment author**=**"Rick Henry"**>**React is such a great framework**!<**/Comment> /**/** MODIFIED

**<**Comment author**=**"Valerie Gibson"**>**I am dreaming **in** React.**<**/Comment> /**/** MODIFIED

**<**/div>

);

}

Let’s now modify our Comment component to render this information. We will be accessing the ‘author’ property, as before, with this.props.author. The text we passed between the <Comment> tags we can access with this.props.children.

*// src/components/Comment.js*

**class** Comment **extends** Component {

render() {

**return** (

**<**div className**=**"comment"**>**

**<**h4**>**{**this**.props.children}**<**/h4> /**/** MODIFIED

**<**p**>**{**this**.props.author}**<**/p> /**/** NEW

**<**/div>

);

}

}

**export** **default** Comment;

Now we can see our comments being rendered to the page.

## Data Model

We have hard coded the data in a list of comments. We now want to create an array of comments which will be dynamically rendered by the CommentList. We are going to set up our CommentBox to be in control of the data. For now it will just store the data, later, when we add a form, it will be responsible for adding new comments to it.

## State

Our CommentBox is going to be the master of the state of our application, the array of comments. For now we’ll just make some mock data. If we were creating a proper app we could get this from an API or database. The state will be stored in the constructor.

*// src/containers/CommentBox.js*

**class** CommentBox **extends** Component {

**constructor**(props) { *// NEW*

**super**(props);

**this**.state **=** {

comments: [

{

id: 1,

author: "Beth Fraser",

text: "I love JS more each day."

},

{

id: 2,

author: "Alan Russell",

text: "Just wait until we add the form. It's gonna be so good."

}

]

};

}

*// ...*

}

And let’s pass the data as props to the CommentList component so it can render it, on the key comments.

**class** CommentBox **extends** Component {

*// ...*

render() {

**return** (

**<**div className**=**"comment-box"**>**

**<**h2**>**Comments**<**/h2>

**<**CommentList comments**=**{**this**.state.comments} /> /**/** UPDATED

**<**/div>

);

}

}

To check this has worked, if you looking in React Dev Tools and click on:

* CommentBox you should see it has the array comments as state.
* CommentList you should see it has the array comments as props.

## Dynamic Rendering of Data

The CommentList has no state, it just has the comments data it has been given by the CommentBox. Inside CommentList we are going to map the array of comment objects (this.props.comments) into an array of Comment components.

*// src/components/CommentList.js*

**class** CommentList **extends** Component {

render() {

**const** commentNodes **=** **this**.props.comments.map(comment **=>** { *// NEW*

**return** (

**<**Comment author**=**{comment.author}**>**{comment.text}**<**/Comment>

);

});

**return** (

**<**div className**=**"comment-list"**>**

**<**Comment author**=**"Rick Henry"**>**React is such a great framework**!<**/Comment>

**<**Comment author**=**"Valerie Gibson"**>**I am dreaming **in** React.**<**/Comment>

**<**/div>

);

}

}

We can now return our new array of Comment components (commentNodes), rather than the hard coded comments.

*// src/components/CommentList.js*

**class** CommentList **extends** Component {

render() {

**const** commentNodes **=** **this**.props.comments.map(comment **=>** {

**return** (

**<**Comment author**=**{comment.author}**>**{comment.text}**<**/Comment>

);

});

**return** (

**<**div className**=**"comment-list"**>**

{commentNodes} *// MODIFIED*

**<**/div>

)

}

}

Note: We can do this because the render function of a React Component must return a DOM element, a collection of DOM elements, or null. In this case we are returning a collection of Components (Comments), which each return DOM elements.

## The key Property

You will see there is a warning in the browser console, asking that you give each item in a list a unique identifier on the key property. This is because React wants to use its virtual DOM to only re-render specific list items when they change, rather than having to re-render the whole list. By giving each component in the array a unique key, we can solve this problem.

*// src/components/CommentList.js*

**class** CommentList **extends** Component {

render() {

**const** commentNodes **=** **this**.props.comments.map(comment **=>** {

**return** (

**<**Comment author**=**{comment.author} key**=**{comment.id}**>** *// MODIFIED*

{comment.text}

**<**/Comment>

);

});

}

*// ...*

}

Great we have created a static application that renders a list of comments. Now we have everything in place to add a form, and use the power of React’s re-rendering to update the page for us, every time a new comment is added.

## Conclusion

We have seen how to plan the structure of our React applications by creating a hierarchy of components that are each responsible for rending one part of the UI. We have also seen how we can pass a collection of data as props, and to render it dynamically.

# React Comments

## Learning Objectives

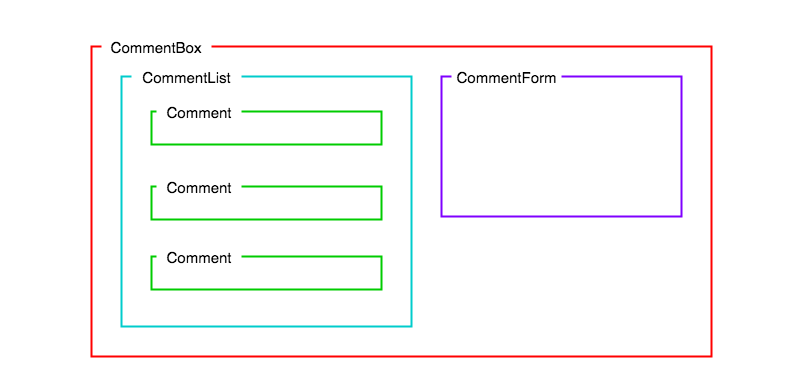
* Create a multiple component React application
* Understand the difference between state and props
* Learn how to pass data between components

## Introduction

We are going to create an application that displays a list of comments and users can add new comments via a form.

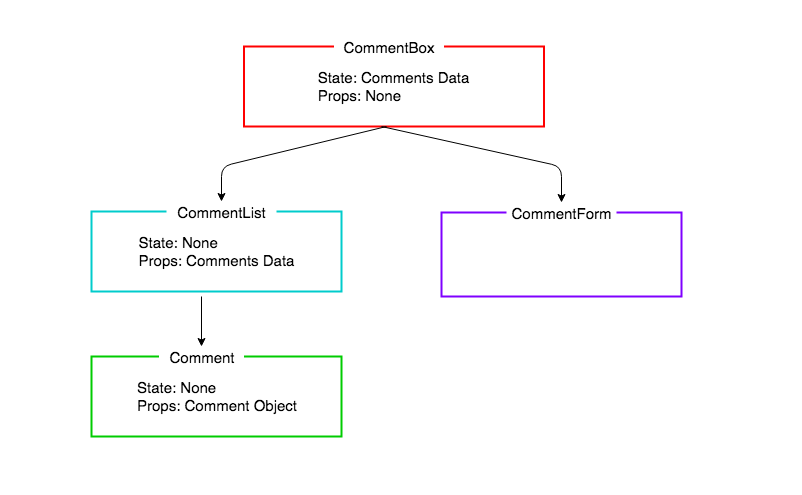
## Application Structure

When planning a React app, we divide up the UI into separate components, with each component responsible for rendering a section of the UI. This is what our component structure is going to look like.



Comments Application Component Structure

CommentsBox will be appended to the HTML root element by React. Then there will be nothing more we need to add to our HTML, the whole application will be drawn by Javascript using React. We do this by describing components for our UI in a component **hierarchy**. Any component rendered by another is called the child of its parent component.



Comments Application Component Hierarchy

In this lesson we will only be concentrating on the left-hand side of the diagram, rendering a list of comments. In the next lesson we will be adding a form, and updating the state, but it’s important to think about the whole application at this point in our planning, because we need to decide which components will have what data for their state and props.

You can see the state and props have already been added to the diagram. The application state is kept high in CommentBox. The reason we keep the state high in our React applications is because of React’s one-way data flow. We do not pass data across the hierarchy chains in React (unlike with other patterns such as PubSub). Therefore, by keeping the comments data at the top level, all the rendering chains can have access to it: the CommentForm can update the data, and the CommentList and Comments can be re-rendered with new comments as they are added to the collection.

As we are not adding the form until the next lesson, in this lesson we will be building a static application that will render based on a hard coded array of comments. This is often a good place to start with a React application.

### Create React App Boilerplate

Let’s use create-react-app to bootstrap our application.

npx create-react-app comments

cd comments

npm start

OK! Through the magic of create-react-app, we’ve pulled down all the bits and pieces we need to start off. We do need to tidy up a little bit, though.

*# cmd-t, new tab*

cd src

rm logo.svg

At this point, the app has broken, because we’ve removed stuff that we’re importing into our app. Let’s fix that. And while we’re at it, let’s remove some default JSX, and render hello world to the page.

**import** React **from** "react";

*// DELETED*

**import** "./App.css";

**function** App() {

**return** (

**<**h1**>**Hello World**!<**/h1> /**/** MODIFIED

);

}

**export** **default** App;

## Containers and Components

We are about to build our first Component, the CommentBox. We can put all our components directly in the ‘src’ folder, but we are going to separate our components into two categories in order to give our application a bit more structure. The two categories are containers and components.

Both will contain React Components, but following convention, if a component contains logic or holds application state, we will put it in the containers directory. If a component just contained presentational code, it will go inside the containers directory.

1. containers - Components with logic or application state
2. components - Components with presentational code

Note: The difference between a container and component is not strictly black and white, but with more experience we will see some more of the differences.

Let’s make those directories:

mkdir src/components

mkdir src/containers

The approach we are going to take it to create the chain of components and check that each one renders the next, before implementing any of the components’ details.

Let’s start with the top of the hierarchy, the CommentBox. Because the CommentBox is going to hold our application state it will go in the containers directory:

touch src/containers/CommentBox.js

*// src/containers/CommentBox.js*

**import** React, { Component } **from** "react";

**class** CommentBox **extends** Component {

render() {

**return** (

**<**div className**=**"comment-box"**>**

Hello, world**!** I am a CommentBox.

**<**/div>

);

}

}

**export** **default** CommentBox;

To get this to render to the page, we need to tell App to render it.

**import** React **from** "react";

**import** CommentBox **from** "./containers/CommentBox"; *// NEW*

**function** App() {

**return** **<**CommentBox **/>** *// MODIFIED*

}

**export** **default** App;

Note: We don’t need the parentheses (()) around the return value, because it is on one line.

Once that is working, we will work our way down the hierarchy, next making the CommentList. The comment list won’t store application state, its sole responsibility will be to be passed the data and render a collection of Comment components.

touch src/components/CommentList.js

*// src/components/CommentList.js*

**import** React, { Component } **from** "react";

**class** CommentList **extends** Component {

render() {

**return** (

**<**div className**=**"comment-list"**>**

Hello, world**!** I am a CommentList.

**<**/div>

);

}

}

**export** **default** CommentList;

For it to be displayed on the page we need to tell our CommentBox component to render it. Let’s also add a title.

*// src/containers/CommentBox.js*

**import** React, { Component } **from** "react";

**import** CommentList **from** "../components/CommentList"; *// NEW*

**class** CommentBox **extends** Component {

render() {

**return** (

**<**div className**=**"comment-box"**>**

**<**h2**>**Comments**<**/h2> /**/** MODIFIED

**<**CommentList **/>** *// NEW*

**<**/div>

);

}

}

**export** **default** CommentBox;

## Comment Component

The last component is the Comment component in the hierarchy and as it is only responsible for receiving data and rendering it to the screen, it will go in the ‘components’ directory.

touch src/components/Comment.js

*// src/components/Comment.js*

**import** React, { Component } **from** "react";

**class** Comment **extends** Component {

render() {

**return** (

**<**div className**=**"comment"**>**

Hello World I am a comment

**<**/div>

);

}

}

**export** **default** Comment;

By the end of this lesson our CommentList is going to dynamically render the collection of comments it’s passed, but for now we are going to hard code a couple of Comment components, to ensure we don’t have any errors in the presentation code.

*// src/components/CommentList.js*

**import** React, { Component } **from** "react";

**import** Comment **from** "./Comment"; *// NEW*

**class** CommentList **extends** Component {

render() {

**return** (

**<**div className**=**"comment-list"**>**

**<**Comment **/>** *// NEW*

**<**Comment **/>** *// NEW*

**<**/div>

);

}

}

**export** **default** CommentList;

We should now see those two comments being rendered to the screen. Next’s let’s populate the comments with come text. Again, by the end of the lesson this content for each comment will be stored in the collection of data, but for now we will hard code the values in, to ensure our Comment component is rendering it properly.

Let’s pass the author and comment to the Comment component as props. We will pass an ‘author’ property as we have been doing with props previously, but the comment text we are going to use a special kind of property, called children. The children props refers to any text or elements passed between the JSX opening and closing tags.

render() {

**return** (

**<**div className**=**"comment-list"**>**

**<**Comment author**=**"Rick Henry"**>**React is such a great framework**!<**/Comment> /**/** MODIFIED

**<**Comment author**=**"Valerie Gibson"**>**I am dreaming **in** React.**<**/Comment> /**/** MODIFIED

**<**/div>

);

}

Let’s now modify our Comment component to render this information. We will be accessing the ‘author’ property, as before, with this.props.author. The text we passed between the <Comment> tags we can access with this.props.children.

*// src/components/Comment.js*

**class** Comment **extends** Component {

render() {

**return** (

**<**div className**=**"comment"**>**

**<**h4**>**{**this**.props.children}**<**/h4> /**/** MODIFIED

**<**p**>**{**this**.props.author}**<**/p> /**/** NEW

**<**/div>

);

}

}

**export** **default** Comment;

Now we can see our comments being rendered to the page.

## Data Model

We have hard coded the data in a list of comments. We now want to create an array of comments which will be dynamically rendered by the CommentList. We are going to set up our CommentBox to be in control of the data. For now it will just store the data, later, when we add a form, it will be responsible for adding new comments to it.

## State

Our CommentBox is going to be the master of the state of our application, the array of comments. For now we’ll just make some mock data. If we were creating a proper app we could get this from an API or database. The state will be stored in the constructor.

*// src/containers/CommentBox.js*

**class** CommentBox **extends** Component {

**constructor**(props) { *// NEW*

**super**(props);

**this**.state **=** {

comments: [

{

id: 1,

author: "Beth Fraser",

text: "I love JS more each day."

},

{

id: 2,

author: "Alan Russell",

text: "Just wait until we add the form. It's gonna be so good."

}

]

};

}

*// ...*

}

And let’s pass the data as props to the CommentList component so it can render it, on the key comments.

**class** CommentBox **extends** Component {

*// ...*

render() {

**return** (

**<**div className**=**"comment-box"**>**

**<**h2**>**Comments**<**/h2>

**<**CommentList comments**=**{**this**.state.comments} /> /**/** UPDATED

**<**/div>

);

}

}

To check this has worked, if you looking in React Dev Tools and click on:

* CommentBox you should see it has the array comments as state.
* CommentList you should see it has the array comments as props.

## Dynamic Rendering of Data

The CommentList has no state, it just has the comments data it has been given by the CommentBox. Inside CommentList we are going to map the array of comment objects (this.props.comments) into an array of Comment components.

*// src/components/CommentList.js*

**class** CommentList **extends** Component {

render() {

**const** commentNodes **=** **this**.props.comments.map(comment **=>** { *// NEW*

**return** (

**<**Comment author**=**{comment.author}**>**{comment.text}**<**/Comment>

);

});

**return** (

**<**div className**=**"comment-list"**>**

**<**Comment author**=**"Rick Henry"**>**React is such a great framework**!<**/Comment>

**<**Comment author**=**"Valerie Gibson"**>**I am dreaming **in** React.**<**/Comment>

**<**/div>

);

}

}

We can now return our new array of Comment components (commentNodes), rather than the hard coded comments.

*// src/components/CommentList.js*

**class** CommentList **extends** Component {

render() {

**const** commentNodes **=** **this**.props.comments.map(comment **=>** {

**return** (

**<**Comment author**=**{comment.author}**>**{comment.text}**<**/Comment>

);

});

**return** (

**<**div className**=**"comment-list"**>**

{commentNodes} *// MODIFIED*

**<**/div>

)

}

}

Note: We can do this because the render function of a React Component must return a DOM element, a collection of DOM elements, or null. In this case we are returning a collection of Components (Comments), which each return DOM elements.

## The key Property

You will see there is a warning in the browser console, asking that you give each item in a list a unique identifier on the key property. This is because React wants to use its virtual DOM to only re-render specific list items when they change, rather than having to re-render the whole list. By giving each component in the array a unique key, we can solve this problem.

*// src/components/CommentList.js*

**class** CommentList **extends** Component {

render() {

**const** commentNodes **=** **this**.props.comments.map(comment **=>** {

**return** (

**<**Comment author**=**{comment.author} key**=**{comment.id}**>** *// MODIFIED*

{comment.text}

**<**/Comment>

);

});

}

*// ...*

}

Great we have created a static application that renders a list of comments. Now we have everything in place to add a form, and use the power of React’s re-rendering to update the page for us, every time a new comment is added.

## Conclusion

We have seen how to plan the structure of our React applications by creating a hierarchy of components that are each responsible for rending one part of the UI. We have also seen how we can pass a collection of data as props, and to render it dynamically

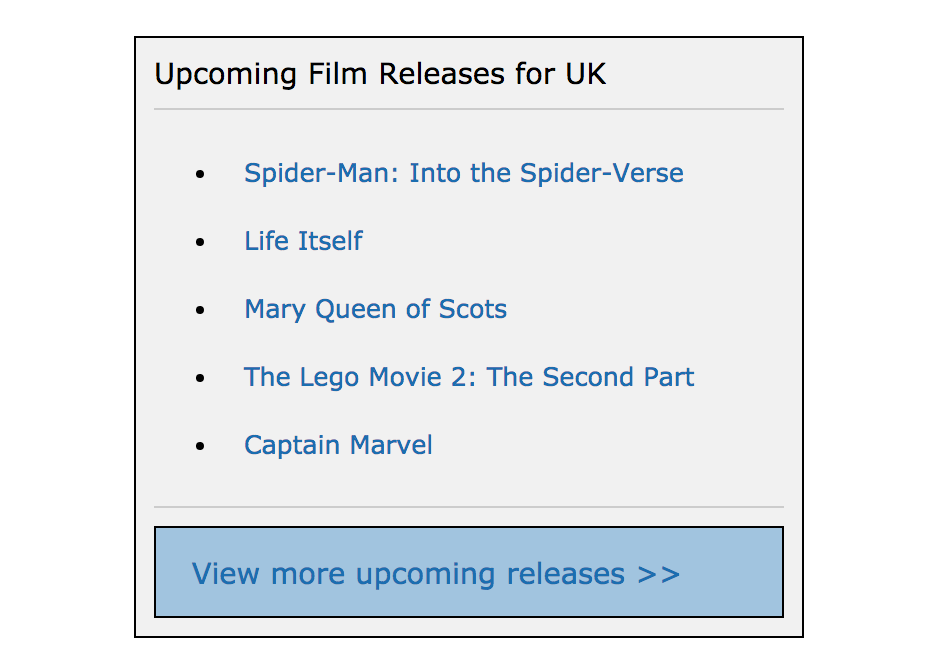
# React: Film Releases

## Reading

[Thinking In React (20 mins)](https://facebook.github.io/react/docs/thinking-in-react.html)

## Practice

You task is to plan and build an application that displays a list of upcoming film titles, each one linking to their [imdb](https://www.imdb.com/) page.

 Screenshot of example result

* Draw out the UI on paper, breaking it down into components.
* Draw a component hierarchy diagram, noting down any state and props for each component.
* Implement the application in React.

Note: The ‘View more upcoming releases’ button can link to the imdb calendar page - <https://www.imdb.com/calendar/?region=gb>

### Data

You can use the following data to display in your application:

[

{

id: 1,

name: "Spider-Man: Into the Spider-Verse",

url: "https://www.imdb.com/title/tt4633694/?ref\_=rlm"

},

{

id: 2,

name: "Life Itself",

url: "https://www.imdb.com/title/tt5989218/?ref\_=rlm"

},

{

id: 3,

name: "Mary Queen of Scots",

url: "https://www.imdb.com/title/tt2328900/?ref\_=rlm"

},

{

id: 4,

name: "The Lego Movie 2: The Second Part", url: "https://www.imdb.com/title/tt3513498/?ref\_=rlm"

},

{

id: 5,

name: "Captain Marvel",

url: "https://www.imdb.com/title/tt4154664/?ref\_=rlm"

}

]

# React Comments Form

## Learning Objectives

* Be able to add a form to a React component
* Be able to pass callbacks between components as props
* Know how and when to trigger a components render method

## Introduction

In the previous lesson we made a static application that renders a list of comments. Now let’s look at how we can update our list with a new comment submitted by a user. To do this we need to alter the state of our application when the new comment gets submitted, causing a rerender.

## Adding new Comments

We are displaying the comments nicely. Now we would like to be able to add comments. Let’s create a form component:

touch src/components/CommentForm.js

Let’s create a form with author and text fields. The JSX syntax is again, very similar to the HTML syntax for a form.

*// src/components/CommentForm.js*

**import** React, { Component } **from** 'react';

**class** CommentForm **extends** Component {

render() {

**return** (

**<**form className**=**"comment-form"**>**

**<**input

type**=**"text"

placeholder**=**"Your name"

**/>**

**<**input

type**=**"text"

placeholder**=**"Say something..."

**/>**

**<**input type**=**"submit" value**=**"Post" **/>**

**<**/form>

)

}

}

**export** **default** CommentForm;

And let’s import CommentForm in CommentBox and tell our CommentBox to render the form

*// src/containers/CommentBox.js*

**import** React, { Component } **from** "react";

**import** CommentList **from** "../components/CommentList";

**import** CommentForm **from** "../components/CommentForm"; *// NEW*

*// ...*

render() {

**return** (

**<**div className**=**"comment-box"**>**

**<**h2**>**Add a Comment**<**/h2> /**/** NEW

**<**CommentForm **/>** *// NEW*

**<**h2**>**Comments**<**/h2>

**<**CommentList comments**=**{**this**.state.comments} />

**<**/div>

)

}

## React Controlled Components

React documentation suggests that when we are creating a component that contains user inputs, such as a form, it should have state that keep’s track of the values of the inputs. This might seem odd, as we have been working to the principle that all the state lives at the top of the application, but this is an exception. The form component will keep track of its own state.

[link to React Controlled Components](https://facebook.github.io/react/docs/forms.html)

The idea is that the form component has state reflects the current state of the inputs, and this state is the single source of truth for the whole component. Anywhere we need to access the values of the form, we access them from the state.

So let’s set that up in the constructor.

*// src/components/CommentForm.js*

**class** CommentForm **extends** Component {

**constructor**(props) { *// NEW*

**super**(props);

**this**.state **=** {

author: '',

text: ''

};

}

*// ...*

}

We can now set up the form to display those state values.

render() {

**<**form className**=**"comment-form"**>**

**<**input

type**=**"text"

placeholder**=**"Your name"

value**=**{**this**.state.author} *// NEW*

/>

**<**input

type**=**"text"

placeholder**=**"Say something..."

value**=**{**this**.state.text} *// NEW*

/>

**<**input type**=**"submit" value**=**"Post" **/>**

**<**/form>

}

**export** **default** CommentForm;

So now when we look at the page, when we type in the fields the values don’t change. This is because they are always displaying the state, which isn’t being updated. Let’s now update the state as the user types.

We will write two functions, one for updating ‘author’ and the other for updating ‘text’. Then we will pass these functions to the onChange event of the inputs. This means as the user types, the inputs’ onChange event will be fired, and the state will be updated.

*// ...*

handleAuthorChange(event) { *// NEW*

**this**.setState({author: event.target.value});

}

handleTextChange(event) { *// NEW*

**this**.setState({text: event.target.value});

}

render() {

**return** (

**<**form className**=**"comment-form"**>**

**<**input

type**=**"text"

placeholder**=**"Your name"

value**=**{**this**.state.author}

onChange**=**{**this**.handleAuthorChange} *// NEW*

/>

**<**input

type**=**"text"

placeholder**=**"Say something..."

value**=**{**this**.state.text}

onChange**=**{**this**.handleTextChange} *// NEW*

/>

**<**input type**=**"submit" value**=**"Post" **/>**

**<**/form>

)

}

If we now check that in out browser, we should see an error. That’s because the handle change functions have the wrong context. So let’s bind them in the constructor to fix this.

*// src/components/CommentForm.js*

**constructor**(props) {

**super**(props);

**this**.state **=** {

author: '',

text: ''

};

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

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

}

Now our form inputs display the value correctly. Next we want to write a function that is called then the submit button it clicked. It’s going to have two responsibilities:

1. Do any form validation, in our case, checking if either field has been left empty
2. Update the list of comments with the new comment
3. Reset the form’s input fields

First let’s do the form validation, accessing the form’s values from the state.

*// src/components/CommentForm.js*

**constructor**() {

*// ...*

}

handleSubmit(event) {

event.preventDefault();

**const** author **=** **this**.state.author.trim();

**const** text **=** **this**.state.text.trim();

**if** (**!**text **||** **!**author) {

**return**

}

}

Next we want to update the state in CommentBox, adding the new comment. We are going to do this last, so for now we will put in a ‘to-do’ comment.

handleSubmit(event) {

event.preventDefault();

**const** author **=** **this**.state.author.trim();

**const** text **=** **this**.state.text.trim();

**if** (**!**text **||** **!**author) {

**return**

}

*// TODO: Update the list of comments*

}

Lastly, we will reset the form fields to be empty, by resetting our form value’s single source of truth: the state.

*// src/components/CommentForm.js*

handleSubmit(event) {

event.preventDefault();

**const** author **=** **this**.state.author.trim();

**const** text **=** **this**.state.text.trim();

**if** (**!**text **||** **!**author) {

**return**

}

*// TODO: Update the list of comments*

**this**.setState({

author: '',

text: ''

});

}

Before we complete the final step of updating the state with the new comment, let’s add this method to the form’s submit event, and check it works.

*// src/components/CommentForm.js*

render() {

**return** (

**<**form className**=**"comment-form" onSubmit**=**{**this**.handleSubmit}**>**

*// ...*

)

}

And let’s bind the handle submit in the constructor.

*// src/components/CommentForm.js*

**constructor**(props) {

*// ...*

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

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

**this**.handleSubmit **=** **this**.handleSubmit.bind(**this**); *// NEW*

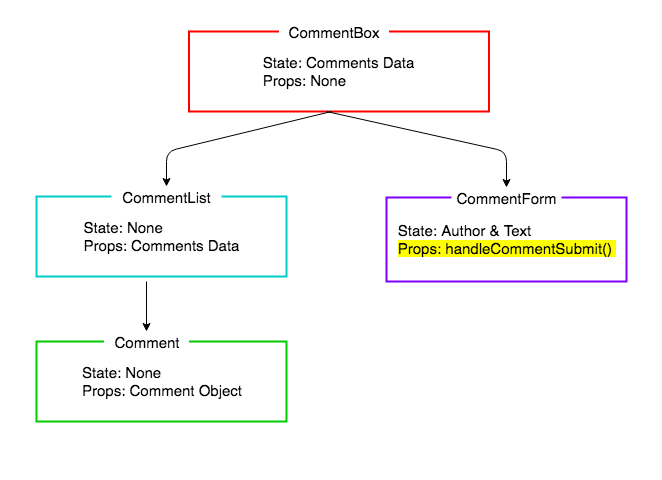
}

Now we should see the form resets every time you submit the form. If we look in React Dev Tools, we should also see the CommentForm’s state being reset each time.

So our final step is to update the CommentForm’s state with the new comment.

## Updating the Comment List

When a user submits a comment, a new comment should be added to our list of our comments. The collection of comment data is being stored in the CommentBox, and we have access to our new comment in CommentForm. So you might be wondering, considering React has a one-way data flow, how can the CommentForm update the state in CommentBox? The answer is that we define a function in Commentbox (handleCommentSubmit) that is responsible for updating the data. Then we pass the function, via props, to CommentForm so that it can invoke it, passing in the new comment. This is just the same as passing any callack in JavaScript, the only different being, we pass it via props from one component to another.



Callback Passed as a Props

When a user submits the form, the state of our application should change and thus the whole page should re-render. As our CommentBox is at the top of the chain this will cause a cascade re-rendering our whole display. It is here we will start to see the benefits of the one-way flow design.

This is going to be a three step process:

1. Define the callback (handleCommentSubmit) in CommentBox. The callback will take a new comment as an argument, and add it to the state.
2. Pass the callback (handleCommentSubmit) via props to CommentForm.
3. Call the callback, passing in the new comment object.

Step 1: defining the callback.

The callback will take a new comment as an argument, and add it to the state.

As we always change state by calling setState, as can’t just push the new comment onto the array in state. We have to create a new array, modify it, then call setState. Here we are using the spread operator. This is another new es6 feature which is perfect for just this case; is separates all the elements of one array into a new array. We then add the new comment to the new array at the same time.

*// src/containers/CommentBox.js*

**constructor**() {

*// ...*

}

handleCommentSubmit(submittedComment) {

submittedComment.id **=** Date.now();

**const** updatedComments **=** [...**this**.state.comments, submittedComment];

**this**.setState({

comments: updatedComments

});

}

Note: The comments need an id as we are using the for the key property on the Comment components. They need to be unique, so for now we will use the date. We should use a hashing function or id generated by a database.

Step 2: Passing the callback as props.

render() {

**return** (

**<**div className**=**"comment-box"**>**

**<**h1**>**Comments**<**/h1>

**<**CommentForm onCommentSubmit**=**{**this**.handleCommentSubmit} /> /**/** UPDATED

**<**CommentList comments**=**{**this**.state.comments} />

**<**/div>

)

}

And don’t forget to bind handleCommentSubmit in the constructor.

*// src/containers/CommentBox.js*

**constructor**(props) {

*// ...*

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

}

Step 3: Invoking the callback, passing in the new comment object.

Now we want the CommentForm to call this function when a comment is submitted.

*// src/components/CommentForm.js*

handleSubmit(event) {

event.preventDefault();

**const** author **=** **this**.state.author.trim();

**const** text **=** **this**.state.text.trim();

**if** (**!**text **||** **!**author) {

**return**

}

**this**.props.onCommentSubmit({

author: author,

text: text

}); *// MODIFIED*

**this**.setState({

author: '',

text: ''

});

}

Fantastic we have our application dynamically updating using the React one way flow. Now when a user submits the form, the list of comments it updated on the page.

## Conclusion

Updating state in the top level component by a component further down the chain is done by passing a callback as props. When the callback is invoked, it sets state, and a render is called, re-rendering the page with the new information.

We have also seen how to create Controlled Components for components with inputs, that have a single source of truth for the form values.

# Lab: Stateless Functional Components

**Duration: 30 mins**

## Learning objectives

* understand what a stateless react component is and why we might use one
* learn the syntax for writing a stateless component

## Brief

Your task is to read the information below about stateless functional components and refactor the comments application to use stateless functional components wherever possible.

## Stateless Functional Components

If a React component does not need to have its own internal state and is only rendering data passed down to it through props, there is an alternative syntax that we can use for creating it. This type of component is called a stateless component or stateless function.

### Benefits

There are a few reasons why we might use a stateless component. Firstly they usually take fewer lines of code to write so are both quicker to create and easier to read. Win!

Another reason is that because you can’t use state in one, it makes you think more carefully about what you’re putting into your React components and how you should stucture your app. Since the purpose of React is to keep all the state near the top of your component tree and have data flow down, any larger app will have plenty of components that only render data passed as props - most of these can be written as stateless components and doing so prevents you from adding unnecessary state too far down the hierarchy.

Finally, the React developers are aiming to make stateless components more efficient than full ones in the future by eliminating extra checks for lifecycle methods etc, so they’re good to know about for the future if you’re building a big app that needs to be optimized for performance.

### Syntax

Instead of using the extends React.Component syntax to make our component, we can use a plain JavaScript function. This takes in one argument, which is the props object. React will pass our stateless function component the props as an argument by default.

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

}

Then instead of using the render method, whatever we return from the function is rendered as our component. We can wrap this in brackets () to write it over multiple lines, using JSX as usual. Anything we want from props we can just get from the props parameter.

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

**return** (

**<**h1**>**Hello {props.name}**<**/h1>

);

}

This also is the perfect opportunity to use the es6 object destructuring feature.

**const** MyComponent **=** ({name}) **=>** {

**return** (

**<**h1**>**Hello {name}**<**/h1>

);

}

We can also use the arrow functions implicit return to make the syntax more concise.

**const** MyComponent **=** ({name}) **=>** **<**h1**>**Hello {name}**<**/h1>

Any extra functionality we need, like event listeners or functions to compute the props before displaying them, you just include above the return like in any other function.

Note: We will still have to import React from ‘react’ into our stateless functional components because JSX is using React methods to create elements and manipulate the DOM.

## Additional Resources

Documentation:

https://facebook.github.io/react/docs/reusable-components.html#stateless-functions

Good article on the benefits of using stateless components:

https://medium.com/@housecor/react-stateless-functional-components-nine-wins-you-might-have-overlooked-997b0d933dbc#.1ts0vetzf

# React Lifecycle

## Learning Objectives

* Understand that React has lifecycle methods
* Look at a few of the methods in action

React automatically looks for functions throughout the life of a component.

We can think of a component as having a birth, life and death, just like us!

You won’t be using all of these but they are handy to know, especially for getting AJAX data.

We are not required to implement these ‘lifecycle’ methods but if we choose to, React will run them automatically for us in the order of the lifecycle.

Think of them like hooks that we can use to run some code if need be.

Open up piggy bank / counter app from first day. Add the following methods in to the main component and see how the lifecycle methods work within it.

## Looking at the lifecycle

Firstly add a console.log to the render method. render is called every time the component re-renders.

Next let’s look at where componentDidMount and componentWillMount are called, and the difference between them:

*// ./src/PiggyBank.jsx*

componentWillMount() {

console.log('Component WILL MOUNT!');

**var** button **=** document.querySelector('button');

console.log("Button:", button);

}

componentDidMount() {

console.log('Component DID MOUNT!');

**var** button **=** document.querySelector('button');

console.log("Button:", button);

}

As we can see, before the component has mounted we don’t have access to any of the DOM elements created by the component.

Two more of the lifecycle methods we can look at are:

*// ./src/components/PiggyBank.js*

componentWillUpdate(nextProps, nextState) {

console.log('Component WILL UPDATE!');

console.log("Next State:", nextState);

console.log("Next Props:", nextProps);

}

componentDidUpdate(prevProps, prevState) {

console.log('Component DID UPDATE!')

console.log("Previous State:", prevState)

console.log("Previous Props:", prevProps)

}

These allow us to look at the props and state before and after the component has rendered.

They don’t get called on the initial render. There is a similar method called componentWillReceiveProps() that gets called when a child component is passed new props from its parent.

Similarly we can access the nextProps and do any changes to state etc that are needed.

## Resources

* Dan Abramov’s lifecycle diagram.

A screenshot of a cell phone

Description automatically generated

* More lifecycle overview:

<http://buildwithreact.com/article/component-lifecycle> <http://busypeoples.github.io/post/react-component-lifecycle/>

* Details of all methods (official docs):

<https://facebook.github.io/react/docs/component-specs.html>

# 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.

# Homework

## Learning objectives

* Practise making a request in a React app and passing the data from that request down the chain of components
* Practise using stateless components

## Task

Make a simple React app that displays the current UK top 20 songs. You can get the data for this here:

https://itunes.apple.com/gb/rss/topsongs/limit=20/json

You are required to use a stateless component as part of your app.

The MVP is to request the data, then display the list of songs with each song showing its chart position, title and artist. If you don’t like music (why?!) you can go to the link below and choose a different iTunes chart for books/movies etc.

## Extensions

Add more information - you could include images or the preview song audio clips. Style this nicely with CSS.

AND/OR

Add a dropdown to choose a genre of music to display - more URLS can be found at this page, just change the ‘xml’ part to ‘json’: https://rss.itunes.apple.com/us/?urlDesc=%2Fgenerator

# React API Request Lab

Practise what we’ve learned in the past few days by making a small React app. This could be similar to the countries app we made that has a dropdown menu which when selected displays some information about that option. You could also select from a list of api’s below or be creative and make something of your choice

Remember to have a container and component folder. Containers will handle data requests, state changes and callbacks. Components will handle visualisation logic (and only have access to props).

Sample APIs:

## Pokemon API

http://pokeapi.co/

E.g. make a dropdown of the original Pokemon which you can select to see their details. This API uses URLS to link to more information rather than having one huge JSON response, so you will need to do a second AJAX request when an option is selected.

To get more than 20 Pokemon returned at once you can add a limit, e.g. http://pokeapi.co/api/v2/pokemon/?limit=151

## Harry Potter API

http://hp-api.herokuapp.com/

E.g. List all of the characters and click to see more info. You could extend this by filtering by Hogwarts House.

## Football Data

https://www.football-data.org/documentation/quickstart

E.g. make a drop down box of all of the teams playing in the Premier League, and when selected show a list of the players for that team with their position and shirt number - this will need a second AJAX request on click. OR Make an app to look at the different football league tables.

You need to sign up for a free API key (http://api.football-data.org/register) which should be emailed to you right away, and then include this in any requests to the API as follows:

**const** url **=** 'http://api.football-data.org/v2/competitions';

fetch(url, {

headers: {

'X-Auth-Token': 'TOKEN GOES HERE'

}

})

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

.then(data **=>** console.log(data))

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

## Learning Objectives

* Know the benefits of React Router
* Be able to use React Router in a simple application

## Introduction

Traditional server rendered applications work by entering the URL into the browser address bar, making a new request for each page. This means that the information being displayed and the URL are kept in sync. Single-Page Applications make the requests asynchronously, therefore the URL gets out of sync with the UI.

This makes navigation difficult. Browser navigation buttons don’t work, and we can’t direct users directly to specific pages of the app. React has a router module which will help us out with this. It allows us to create a Single-Page application where the UI and URL will be kept in sync so we can use the Browser Navigation features.

Let’s first look at an application without React Router and all the problems we mentioned in action.

### Setup

Let’s set up an application.

Instructor note: Hand out the router start point

Instructor note: Ask the class…

Run the start point and check that the links work; that clicking on them renders different the appropriate component.

This is a simple application where the Main component is responsible for storing in its state the current page that is being displayed and for rendering the appropriate view when this changes.

### Task: (5 minutes)

Add a component and a link for a contact page.

# Adding Router

So the application is functional, but the url does not match what the UI is showing. And if we leave or refresh the page we will always go to home page regardless where we were. Enter the need for a router.

npm install react-router-dom

Let’s set up the Main component to work as router. We will import our other components so we can later pass them in our routes. We will also import BrowserRouter (which we rename Router for convenience) from the react-router-dom library.

*// /src/components/Main.js*

**import** React, { Component } **from** "react";

**import** About **from** "./About";

**import** Home **from** "./Home";

**import** Pricing **from** "./Pricing";

**import** { BrowserRouter **as** Router } **from** "react-router-dom"; *//NEW*

And we use this Router component in our render method:

*// /src/components/Main.js*

render() {

**return** (

**<**Router**>**

**<**/Router>

);

}

Router will keep track of our navigation through a site. This means now when we refresh it will remember where we were and we can use our browser navigation buttons to go back and forth through our browsing history.

Now we are going to define our routes inside our Router. We have to wrap them in a React.Fragment because BrowserRouter only expects one element. Each Route takes two attributes, a path and a component to render on that path.

*// /src/components/Main.js*

**import** { BrowserRouter **as** Router, Route } **from** "react-router-dom"; *// UPDATED*

**class** Main **extends** Component {

*// AS BEFORE*

render() {

**return** (

**<**Router**>**

**<**React.Fragment**>**

**<**Route exact path**=**"/" component**=**{Home} />

**<**Route path**=**"/about" component**=**{About} />

**<**Route path**=**"/pricing" component**=**{Pricing} />

**<**/React.Fragment>

**<**/Router>

);

}

}

We set up an an **exact** path to render the Home component. This will render the Home component on http://localhost:3000/ only. The other paths will load their relevant components.

Now we can delete all the methods that were being used to keep track of and render the appropriate component, as Router is now doing all that for us. Let’s delete:

* The methods whose names start with goto...
* The bindings for these methods in constructor
* The pageComponent method
* Actually, the whole constructor!

## Navigation

We can now check the routes we created by visiting them in our browser:

* http://localhost:3000/
* http://localhost:3000/about
* http://localhost:3000/pricing

But typing addresses into the browser is no good. Our app needs a navbar!

touch src/components/NavBar.js

*// /src/components/Main.jsx*

**import** NavBar **from** "./NavBar";

**<**Router**>**

**<**React.Fragment**>**

*// NEW*

**<**NavBar **/>**

{*/\* Routes as before \*/*}

**<**/React.Fragment>

**<**/Router>

Our NavBar is going to be a stateless functional component, and we’re going to make an unordered list of links. In HTML, we would use a tags for links, with href attributes that point to a URL. React Router has a built-in Link component, which has a to attribute to point to its routes. This makes for some really cute code:

*// /src/components/NavBar.jsx*

**import** React **from** "react";

**import** { Link } **from** "react-router-dom";

**const** NavBar **=** () **=>** (

**<**ul**>**

**<**li**>**

**<**Link to**=**"/"**>**Home**<**/Link>

**<**/li>

**<**li**>**

**<**Link to**=**"/about"**>**About**<**/Link>

**<**/li>

**<**li**>**

**<**Link to**=**"/pricing"**>**Pricing**<**/Link>

**<**/li>

**<**/ul>

);

**export** **default** NavBar;

We can see as we navigate through the site, and the URL changes. So our browser’s back button works as we’d expect, and we can bookmark pages in our app.

### Task: (5 minutes)

Create a new component and add it to the list of links in the Main component.

## Passing down props

Loading our components in this way is ideal, we just pass the component to a Route and React Router knows what to do. It knows how to render it, and when. This is without a doubt React Router’s preferred way of linking URL paths with components.

But there is a problem. With this syntax, there is no way of passing props to a component that we have set up a route for. Most of the time, this is fine; our routes point to top-level components, which act like separate pages and don’t need to take in props. But sometimes we do need to pass props down, and luckily React Router has a very nice syntax for this.

Let’s say we wanted to pass some actual pricing data to our Pricing page component. Maybe in the real world this data might come from our server, but for now we’ll just hard-code it in our Main component’s state:

*// /src/components/Main.js*

**class** Main **extends** Component {

**constructor**(props) { *// NEW*

**super**(props);

**this**.state **=** {

pricing: [

{level: "Hobby", cost: 0},

{level: "Startup", cost: 10},

{level: "Enterprise", cost: 100}

]

};

}

render() {

*// AS BEFORE*

}

}

Right now we’re passing our Pricing component into a Route through the Route’s component property. This is fine for most React Router routes, but there’s no way to pass props into Pricing now. Instead, we have to tell the Route exactly what to render here, through its render prop:

**class** Main **extends** Component {

render() {

**return** (

**<**Router**>**

**<**React.Fragment**>**

*// ... AS BEFORE*

**<**Route *// UPDATED HERE*

path**=**"/pricing"

render**=**{() **=>** **<**Pricing prices**=**{**this**.state.pricing} />}

**/>**

**<**/React.Fragment>

**<**/Router>

);

}

}

We can check this out in React dev tools and see that when the Pricing component mounts at http://localhost:3000/pricing it gets passed an array of 3 prices as props.

We can make use of these props to render a price list:

*// /src/components/Pricing.js*

**const** Pricing **=** ({prices}) **=>** { *// UPDATED*

**const** listItems **=** prices.map((price, index) **=>** { *// NEW*

**return** **<**li key**=**{index}**>**{price.level}: £{price.cost} per month**<**/li>

})

**return** (

**<**div**>**

**<**h4**>**Pricing**<**/h4>

**<**ul**>**

{ listItems } *// UPDATED*

**<**/ul>

**<**/div>

)

};

## Switch

Lastly, let’s look at what would happen if we wanted to add a default component that renders in the event of a user trying to hit a route that we haven’t defined. Let’s create an ErrorPage component.

touch src/components/ErrorPage.js

*// /src/components/ErrorPage.js*

**const** ErrorPage **=** ()**=>** (

**<**h1**>**404 **-** PAGE NOT FOUND**<**/h1>

)

**export** **default** ErrorPage

And then let’s implement it in our Router:

*// /src/components/Main.js*

**import** ErrorPage **from** './ErrorPage.js' *//NEW*

**<**Router**>**

**<**React.Fragment**>**

**<**NavBar **/>**

**<**Route exact path**=**"/" component**=**{Home} />

**<**Route path**=**"/about" component**=**{About} />

**<**Route path**=**"/pricing" component**=**{Pricing} />

**<**Route component**=**{ErrorPage} /> /**/**NEW

**<**/React.Fragment>

**<**/Router>

If we look in the browser now, we can see an issue - our ErrorPage component is rendering on every page of our app. This isn’t what we wanted to happen! The way we can solve this is by bringing in the Switch component that’s also included with react-router-dom.

By implementing a switch component and wrapping our routes inside it, we can ensure that the only the component we want displayed is rendered when the browser hits our routes. Like so:

*// /src/components/Main.js*

**import** { BrowserRouter **as** Router, Route, Switch } **from** "react-router-dom"; *// UPDATED*

**<**Router**>**

**<**React.Fragment**>**

**<**NavBar **/>**

**<**Switch**>**

**<**Route exact path**=**"/" component**=**{Home} />

**<**Route path**=**"/about" component**=**{About} />

**<**Route path**=**"/pricing" component**=**{Pricing} />

**<**Route component**=**{ErrorPage} />

**<**/Switch>

**<**/React.Fragment>

**<**/Router>

Now if we check back, everything should be working as expected. "/", "/about", "/pricing" all display their respective components, and any other path we type into our browser address bar will give us our ErrorPage component.

How is switch doing this? Before, we would see every Route that matched our path - "/about" would display our About component, our ErrorPage component, and say hypothetically if we had a Route rendering a component with a path that held a parameter variable, "/:id" for instance, that too would be rendered. Prior to our switch, router was rendering components inclusively - anything that could feasibly match the same path was appearing. Now, we mitigate that with switch, which achieves this by rendering the first route that matches the path we’re trying to reach in our browser exclusively.

There is a pitfall there however - if the default Route was the first one in our Router, that’s the only one that will ever display, because the switch sees that it matches, renders it and stops. So just as we have in other programming languages, if we’re implementing a switch, we need to be mindful of the ordering of our Routes.

## Summary

In this lesson we’ve seen how React Router enables our users to use standard browser features like a Back button and bookmarks, by matching our UI to appropriate URLs. We can set up Routes within a Router component, match them to specific URL paths, and load components when the browser requests those routes. We can also pass properties into the components that React Router loads. We have also seen how to use the Link component in place of a tags to take advantage of React Router’s features, and how we can use Switch to ensure more control over what components are rendering dependent on the URL path.

**React Requests - Pirate API**

**Learning Objectives**

* Understand how to make a call to our Pirate API using React.
* Understand how to write custom routes to get back the data we want.

**Duration**

1 hour 30 minutes

**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 our own Pirate 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.

**Rest APIs using HAETOS**

Do you remember what Spring Data Rest gave us? It gave us HAETOS route that adhered to the 7 restful routes. The good news is that we can easily use these routes to connect to our own API.

If you haven’t already, go and grab the JSON formatter extension for Chrome:

https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?hl=en

This will make it a lot easier to read our JSON results.

Let’s have a look at the api we are going to use today - the PirateService api.

Hand out start point and have students. Run the PirateService in IntelliJ and keep running for all lessons in this series.

As you can see this back end is the same as we created last week with projections set for all 3 models to get back all the data associated with each.

**Task**

Run the back-end application in IntelliJ. Visit the routes in your browser and look at the data.

So today we will be using 2 different types of URL in our apps.

We will be using /pirates in React to get back our list of pirates but also /pirates to hit our API

SO React will use http://localhost:3000/pirates and Spring will use http://localhost:8080/pirates to hit the API. Yuck! This could get really really confusing. So let’s make it clearer by changing the base route for Spring so that we need to hit /api/pirates to get back the data from Spring.

In the Java code open application.properties and add the following line.

*<!-- application.properties -->*

server.servlet.context-path=/api

Now visit the browser and go to http://localhost:8080/api/pirates.

Great we now have our data back on this route so we will use that moving forward for whatever data we want. So if we wanted all ships it would be /api/ships etc…

So the main route we are going to use today is this one

http://localhost:8080/api/pirates

This gives us ALL the pirate data.

Let’s have a go.

**React request**

Have students open front end code and look at it in atom. Note that we have a NavBar, route and css included.

Run npm install and npm start to kick off the server.

Here we have a React application set up with a main controller. It is in here that we will set up routes to the top level components for pirates, ships and raids.

Let’s add a a PirateContainer which will be the top level of our Pirates component tree.

touch src/containers/PirateContainer.js

Next we will add a route to this in MainContainer.

*// MainContainer.js*

**import** PirateContainer **from** './PirateContainer'; *// ADDED*

*//AS BEFORE*

**const** MainContainer **=** () **=>** {

**return** (

**<**Router**>**

**<**React.Fragment**>**

**<**NavBar**/>**

**<**Switch**>**

**<**Route path**=**"/pirates" component**=**{PirateContainer}/>

**<**/Switch>

**<**/React.Fragment>

**<**/Router>

)

}

Ok so we will be able to code up our controller and when we go to /pirates it will launch into this.

Our Pirate container will deal with the pirate specific routes. We will start with all pirates so will need this route to render our PirateList component. We don’t need to pass a path to this route as this will be our default.

Let’s code up the PirateContainer.

*// PirateContainer.js*

**import** React, {Component, Fragment} **from** 'react';

**import** {BrowserRouter **as** Router, Route, Switch} **from** 'react-router-dom';

**import** PirateList **from** '../components/pirates/PirateList';

**class** PirateContainer **extends** Component {

**constructor**(props){

**super**(props);

}

render(){

**return**(

**<**Router**>**

**<**Fragment**>**

**<**Switch**>**

**<**Route render**=**{(props) **=>** {

**return** **<**PirateList **/>**

}} />

**<**/Switch>

**<**/Fragment>

**<**/Router>

)

}

}

**export** **default** PirateContainer;

Now if we look at the browser and go to localhost:3000/pirates we should see the text I am a list of pirates.

**Requests**

So our PirateList is going to need a list of pirates.

To get this we will be making a request to our API

As we will no doubt be making multiple requests to get data we can abstract out the method to a helper. Just like we did in Vanilla JS. We have set up a request helper and coded in the get method like we had before.

Open request.js in the helpers folder.

*//request.js*

**class** Request {

**get**(url) {

**return** fetch(url)

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

} *// ADDED*

}

**export** **default** Request;

And import this into PirateContainer to use.

*//PirateContainer.js*

**import** React, {Component} **from** 'react';

**import** {BrowserRouter **as** Router, Route, Switch} **from** 'react-router-dom';

**import** PirateList **from** '../components/pirates/PirateList';

**import** Request **from** '../helpers/request'; *// ADDED*

Now in our PirateContainer we will fetch the data in componentDidMount and we will log out the data we get back initially.

Let’s start by setting up the state in the constructor.

*// PirateContainer.js*

**constructor**(props){

**super**(props);

**this**.state **=** {

pirates: []

} *// ADDED*

}

Next we will add componentDidMount method and set up our request.

*//PirateContainer.js*

**class** PirateContainer **extends** Component {

**constructor**(props){

**super**(props);

**this**.state **=** {

pirates: []

}

}

componentDidMount(){

**const** request **=** **new** Request();

request.**get**('/api/pirates')

.then((data) **=>** {

console.log(data)

})

}

}

**Proxy**

Next thing to deal with is when we make a fetch request in React the request is actually sent to localhost:3000/api/pirates.

We don’t want this… we want to hit localhost:8080/api/pirates.

We could pass in the full url into the get but then this would give us CORS errors that we would need to deal with in our Spring app.

A better way around this is to assign a proxy value in our package.json

With the proxy any requests that are sent out of the application will use the proxy rather than reacts. So if we set the proxy to http://localhost:8080/ any fetch request will use the Spring server.

So our fetch /pirates will actually resolve to http://localhost:8080/api/pirates.

**"browserslist"**: [

">0.2%",

"not dead",

"not ie <= 11",

"not op\_mini all"

],

**"proxy"**: "http://localhost:8080/" \*/ ADDED /\*

Note you may have to restart the server for this to take effect.

Cool so now we are able to get the data back.

If we look at the data we see an array with our pirates embedded in the results.

So now all we need to do is set the state.

*// PirateContainer.js*

componentDidMount(){

**const** request **=** **new** Request();

request.**get**('/api/pirates')

.then((data) **=>** {

**this**.setState({pirates: data})

})

}

Let’s amend the route and pass in props to the PirateList.

Render gives us access to props and allows us to pass props to a component. (Just like the normal render method).

*//PirateContainer.js*

**<**Route render**=**{(props) **=>**{

**return** **<**PirateList pirates**=**{**this**.state.pirates}/>

}}/>

In PirateList we can now use our React DevTools and check that we have the array of pirates as props.

**Checking the data**

As the request may take some time we only want to start creating our Pirate components when we have data.

We will put in a check at top of PirateList to make sure we have it.

*// PirateList.js*

**if** (props.pirates.length **===** 0){

**return** (**<**p**>**Loading...**<**/p>)

}

**return** (

**<**div**>**Im a list **of** pirates**!<**/div>

)

Cool we can now see the pirates array being logged out from within PirateList.

We will change the div and instead we will use a ul to display a list of Pirate components.

So first we want to create an li tag for each Pirate. To do this we want to loop over the array of pirates and return a collection of li tags with a Pirate component in each. We also want each individual Pirate object passed into them in turn.

We can use Array.map to achieve this.

We will also add a surrounding div for CSS purposes. The div will have a className of component to apply the CSS.

*//PirateList.js*

**const** pirates **=** props.pirates.map((pirate, index) **=>** {

**return** (

**<**li key**=**{index} className**=**"component-item"**>**

**<**div className**=**"component"**>**

**<**Pirate pirate**=**{pirate} />

**<**/div>

**<**/li>

)

}) *// ADDED*

**return** (

**<**div**>**Im a list **of** pirates**!<**/div>

)

Next we will change the div to a ul and render the li elements inside.

*//PirateList.js*

**return** (

**<**ul className**=**"component-list"**>**

{pirates}

**<**/ul>

) *// MODIFIED*

Cool so we should now see a list of divs with I am a pirate displayed.

Let’s change this to display the Pirates details. We will put this in a React Fragment for something we will use later on.

*//Pirate.js*

**import** React, {Fragment} **from** 'react';

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

**if** (**!**props.pirate){

**return** "Loading..."

}

**return** (

**<**Fragment**>**

**<**p**>**

{props.pirate.firstName} {props.pirate.lastName}

**<**/p>

**<**p**>**Age: {props.pirate.age}**<**/p>

**<**p**>**Ship: {props.pirate.ship.name}**<**/p>

**<**/Fragment>

)

}

**export** **default** Pirate;

Fantastic we now have a list of pirates showing all their details.

**Task**

* Fetch all Ships and raids.

**Summary**

* Learned how to make a call to our Pirate API using React.

**Next Lesson**

* Display a single pirate details

# React Requests - Single Pirate from API

## Learning Objectives

* Understand how to make a call to our Pirate API using React to fetch a single Pirate.
* Understand how to use built in props in routes to pass Id as prop.

## Duration

1 hour.

# Intro

So we have been able to get all of the Pirates from our API but we may also want to display more details about a single pirate. Well we can define a route using the Pirates ID to display a single pirate and create a different component for additional pirate details.

# Routes.

Use end point from all pirates lesson to start this lesson.

So we will need another route to get our single Pirate. This route will follow the pattern /pirates/1. Where 1 is the pirates id.

As the id will be different each time we will need to use params to get the ID from the url. The url params are accessible using the props passed into render. The pattern for this will be /pirates/:id.

Let’s start by defining the route in our PirateContainer. So we are sure no other routes will match we make the path exact.

(Our route to all pirates is default so this will always be the last route.)

*// PirateContainer.js*

**<**Router**>**

**<**Fragment**>**

**<**Switch**>**

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

}}/>

**<**Route render**=**{(props) **=>**{

**return** **<**PirateList pirates**=**{**this**.state.pirates}/>

}}/>

**<**/Switch>

**<**/Fragment>

**<**/Router>

To get the id from the url we use props.match.params.id. This will bring back the id as a string.

*// PirateContainer.js*

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** id **=** props.match.params.id;

}

}/>

Next we need to find the pirate with that id. We already have all the pirates in our state so we will write a function above to find the pirate by id. We will then use this in our route to get the pirate.

As the id from the params is a string we will also need to change it to an int in our find function.

*// PirateContainer.js*

componentDidMount(){

*// AS BEFORE*

}

findPirateById(id){

**return** **this**.state.pirates.find((pirate) **=>** {

**return** pirate.id **===** parseInt(id);

});

} *// ADDED*

render(){

*// AS BEFORE*

**<**Route exact path**=**"/pirates" render**=**{(props) **=>**{

**return** **<**PirateList pirates**=**{**this**.state.pirates}/>

}}/>

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** id **=** props.match.params.id;

**const** pirate **=** **this**.findPirateById(id); *// Added*

}

} />

}

And bind the new method in the constructor.

*// PirateContainer.js*

**constructor**(props){

**super**(props);

**this**.state **=** {

pirates: []

}

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

}

When we get back a single pirate we probably want to show more details. Like maybe the list of raids that the pirate has been on.

We will create a new component called PirateDetail for this and pass the Pirate as props.

We will also import it just now even though this will give us an error until we have created the component.

*// PirateContainer.js*

**import** PirateDetail **from** '../components/pirates/PirateDetail'

*// AS BEFORE*

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** id **=** props.match.params.id;

**const** pirate **=** **this**.findPirateById(id);

**return** **<**PirateDetail pirate**=**{pirate}/>

}}/>

Now we will create the PirateDetail component.

touch src/components/pirates/PirateDetail.js

And code this up. We will bring in the basic Pirate component and render this then add details below. We will also make sure the passed in pirate isn’t null. If it is we will return Loading...

As we want to show the list of raids for the pirate we will map over the pirates raids from the API and create an li tag for each one.

We will then render this in a ul below the pirate component.

We will also make this a class for some things that we will be adding later on.

*// PirateDetail.js*

**import** React, {Component} **from** 'react';

**import** Pirate **from** "./Pirate"

**class** PirateDetail **extends** Component {

render(){

**if** (**!this**.props.pirate){

**return** "Loading..."

}

**const** raids **=** **this**.props.pirate.raids.map((raid, index) **=>** {

**return** **<**li key**=**{index}**>**{raid.location}**<**/li>

})

**return** (

**<**div className **=** "component"**>**

**<**Pirate pirate **=** {**this**.props.pirate}/>

**<**p**>**Raids:**<**/p>

**<**ul**>**

{raids}

**<**/ul>

**<**/div>

)

}

}

**export** **default** PirateDetail;

Now we need a Link somewhere to take us to that Route. We need this to go somewhere that we have access to the individual pirate and it’s id.

In Pirate component we will wrap the Pirates name in a Link. We will change the <p> tag surrounding the pirates name to a <Link>.

We also need to define the url outside the return as string concatenation won’t work inside the Link tag.

We will then build the url to hit using the pirates id.

*// Pirate.js*

**if** (**!**props.pirate){

**return** "Loading..."

}

**const** url **=** "/pirates/" **+** props.pirate.id; *// ADDED*

**return** (

**<**Fragment**>**

**<**Link to **=** {url} className**=**"name"**>**

{props.pirate.firstName} {props.pirate.lastName}

**<**/Link> /**/**MODIFIED

**<**p**>**Age: {props.pirate.age}**<**/p>

**<**p**>**Ship: {props.pirate.ship.name}**<**/p>

**<**/Fragment>

)

And import Link at the top of the file

*// Pirate.js*

**import** React, {Fragment} **from** 'react';

**import** {Link} **from** 'react-router-dom'; *// ADDED*

Great so if we refresh our list click on the name of a pirate in our list it should now hit our new route in the container and render the full pirate details.

# Task

* Get one Ship with a list of pirates.
* Get one Raid with a list of pirates.

# Summary

* Learned how to use the data to fetch a pirates details.

#Next Lesson

* Delete a pirate

# React Requests - Delete a Pirate from API

## Learning Objectives

* Understand how to make a call to our Pirate API using React to delete a single Pirate.

## Duration

30 minutes.

# Intro

So far have looked at fetching Pirates from our API but we may also want to update the data. We will look here at making a request to delete a pirate.

In this case we will not set up a new route in React but rather put a delete button on a single Pirate details and use a callback to perform our delete.

# Delete Request.

Use end point from Single Pirate Lesson

Let’s start by adding the delete function in our request helper. Again we will pass in a url to hit.

*// request.js*

**get**(url) {

**return** fetch(url)

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

}

**delete**(url) {

} *// ADDED*

We will still be using fetch here to hit the API but we will declare that the method we want to use is DELETE instead of the default GET. We will also declare that the content type will be JSON in the headers.

These details are passed in as a parameter to the fetch method along with the url. The rest is the same as before so we will return a promise.

*// request.js*

**get**(url) {

**return** fetch(url)

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

}

**delete**(url) {

**return** fetch(url, {

method: "DELETE",

headers: {'Content-Type': 'application/json'}

})

} *// ADDED*

Ok so now we have a delete method set up let’s write a method in PirateContainer to call it. We will pass in the id of the pirate to be deleted.

*// PirateContainer.js*

findPirateById(id){

*// AS BEFORE*

}

handleDelete(id){

**const** request **=** **new** Request()

} *// ADDED*

So the url to delete a pirate from the API will be /api/pirates/{id}. So let’s use the id we passed in to create that url and pass it to our delete method in requests.

*// PirateContainer.js*

handleDelete(id){

**const** request **=** **new** Request();

**const** url **=** '/api/pirates/' **+** id;

request.**delete**(url);

}

After this method runs we probably want to redirect back to the list of pirates. We can set window.location to do this.

*// PirateContainer.js*

handleDelete(id){

**const** request **=** **new** Request();

**const** url **=** '/api/pirates/' **+** id;

request.**delete**(url).then(() **=>** {

window.location **=** '/pirates';

});

}

As we aren’t referencing this in this method we don’t need to bind it.

Lastly we will pass this method to our PirateDetail component as a prop.

*// PirateContainer.js*

{*/\* GET ONE PIRATE \*/*}

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** pirate **=** **this**.findPirateById(props.match.params.id);

**return** **<**PirateDetail pirate**=**{pirate} onDelete**=**{**this**.handleDelete}/> /**/** MODIFIED

}}/>

## Adding a delete button to our pirate.

Now we will add in the method call to PirateDetail component. We will also need to bind this in a constructor

*// PirateDetail.js*

**class** PirateDetail **extends** Component {

**constructor**(props){

**super**(props)

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

}

handleDelete(){

**this**.props.onDelete(**this**.props.pirate.id)

}

*// AS BEFORE*

}

And add the button to our Pirate.

*// SinglePirate.js*

**return** (

**<**div className **=** "component"**>**

**<**Pirate pirate **=** {**this**.props.pirate}/>

**<**p**>**Raids:**<**/p>

**<**ul**>**

{raids}

**<**/ul>

**<**button onClick**=**{**this**.handleDelete}**>**Delete {**this**.props.pirate.firstName}**<**/button> /**/** ADDED

**<**/div>

)

Now when we click the delete button our Pirate should be deleted.

# Summary

* Learned how to make a call to our Pirate API using React to delete single Pirate.

# Next Lesson

* Create a Pirate

# React Requests - Promise.all()

## Learning Objectives

* Understand how to make several requests using Promise.all()

## Duration

30 minutes.

# Intro

So we have been able to get all of the Pirates from our API but we may also want to use all of the ships and raids as well somewhere down the line. (We will be using these when it comes to creating and editing a pirate so we may as well deal with this now. ) Well we can make new requests in our PirateContainer to get all pirates, ships and raids.

As these requests all take time we can make sure that all of these will resolve in succession before we move on. To do this we can use Promise.all() method.

## Promise.all

Promise.all takes in an array of promises. In our case it will be the promises returned from our fetch requests. We can then execute all of the promises in turn and return an array with arrays of the result of each promise.

Let’s add change our PirateContainer, we will create 3 promises one for each of our elements we want to get.

*// PirateContainer*

componentDidMount(){

**const** request **=** **new** Request();

**const** piratePromise **=** request.**get**('/api/pirates'); *// A PROMISE*

**const** shipPromise **=** request.**get**('/api/ships'); *// A PROMISE*

**const** raidPromise **=** request.**get**('/api/raids'); *// A PROMISE*

}

Now we can pass all of these promises to promise.all().

*// PirateContainer*

componentDidMount(){

**const** request **=** **new** Request();

**const** piratePromise **=** request.**get**('/api/pirates'); *// A PROMISE*

**const** shipPromise **=** request.**get**('/api/ships'); *// A PROMISE*

**const** raidPromise **=** request.**get**('/api/raids'); *// A PROMISE*

Promise.all([piratePromise, shipPromise, raidPromise])

}

Promise.all() returns us another promise so we can now use .then() to see our data.

*// PirateContainer*

componentDidMount(){

**const** request **=** **new** Request();

**const** piratePromise **=** request.**get**('/api/pirates'); *// A PROMISE*

**const** shipPromise **=** request.**get**('/api/ships'); *// A PROMISE*

**const** raidPromise **=** request.**get**('/api/raids'); *// A PROMISE*

Promise.all([piratePromise, shipPromise, raidPromise])

.then((data) **=>** {

console.log(data);

})

}

So now we should see an array of arrays. One for pirates, ships and raids.

We can set the state using the index of the arrays in the response.

Add in the ships and raids to the initial state:

*// PirateContainer*

**constructor**(props){

**super**(props);

**this**.state **=** {

pirates: [],

ships:[],

raids: []

}

}

*// PirateContainer*

componentDidMount(){

**const** request **=** **new** Request();

**const** piratePromise **=** request.**get**('/api/pirates'); *// A PROMISE*

**const** shipPromise **=** request.**get**('/api/ships'); *// A PROMISE*

**const** raidPromise **=** request.**get**('/api/raids'); *// A PROMISE*

Promise.all([piratePromise, shipPromise, raidPromise])

.then((data) **=>** {

**this**.setState(

{

pirates: data[0],

ships: data[1],

raids: data[2]

}

)

})

}

And check this in our React DevTools to make sure the state updates.

# Summary

* Learned how to use promise.all() to make several requests to an API.

# Next Lesson

* Edit a pirate.

**React Requests - Add / Remove Raids**

**Learning Objectives**

* Understand how to add or remove raids to a pirate.

**Duration**

60 minutes

**Intro**

So our form works to edit a pirates basic details but we still want the ability to add / remove raids.

Instead of doing this in the form we will add this functionality to our pirate details view.

Next to each raid that pirate is on we will add a delete button.

And we will add a drop down of raids to add that pirate to.

**Delete Raids**

Let’s start by deleting raids.

When we delete a raid from our pirate we will need to update that pirate in the database. So we will pass our handleUpdate method from the container to our pirate details.

*// PirateContainer*

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** id **=** props.match.params.id;

**const** pirate **=** **this**.findPirateById(id);

**return** **<**PirateDetail pirate**=**{pirate}

onDelete**=**{**this**.handleDelete}

onUpdate**=**{**this**.handleUpdate} *// ADDED*

/>

}}/>

Now in the PirateDetail we will add a delete button to our list of raids.

*// PirateDetail.js*

**const** raids **=** **this**.props.pirate.raids.map((raid, index) **=>** {

**return** **<**li key**=**{index}**>**

{raid.location} **<**button**>**Delete**<**/button>

**<**/li>

}) *// AMENDED*

Ok we can see this button and when we click nothing should happen yet.

When we click this button we will need to remove the matching raid in the pirates list of raids.

In order to do this we will need to pass the raid index to a method and use splice to remove from our pirates raid. Then call the update method we passed in as a prop.

We will write the method first. Remember to bind in the constructor!

*// PirateDetail*

**constructor**(props){

*// AS BEFORE*

**this**.deleteRaid **=** **this**.deleteRaid.bind(**this**)

}

handleDelete(){

**this**.props.onDelete(**this**.props.pirate.id)

}

deleteRaid(raidIndex) {

**this**.props.pirate.raids.splice(raidIndex, 1)

**this**.props.onUpdate(**this**.props.pirate)

}

Lastly set our button to trigger this method on click. In order to pass the index as a callback we can’t simply write something like

**<**button onClick**=**{**this**.deleteRaid(index)} **>**

As this would trigger write away.

What we **can** do is pass it into a callback for onClick. Like this

onClick**=**{() **=>** **this**.deleteRaid(index)}

Let’s add that to the button.

*// PirateDetail*

**const** raids **=** **this**.props.pirate.raids.map((raid, index) **=>** {

**return** **<**li key**=**{index}**>**

{raid.location} **<**button onClick**=**{() **=>** **this**.deleteRaid(index)}**>**Delete**<**/button> /**/** AMENDED

**<**/li>

})

Awesome We can now delete the raids attached to our pirate.

Let’s add the ability to give them some more.

**Adding Raids**

We can add a drop down to our pirate to list the raids they don’t already have.

Let’s start by writing a method that checks if a pirate already has a raid. We can use the arrays some method to do this easily.

*// PirateDetail*

deleteRaid(raidIndex) {

*// AS BEFORE*

}

pirateHasRaid(raid){

**return** **this**.props.pirate.raids.some((pirateRaid) **=>** {

**return** raid.id **===** pirateRaid.id

})

}

And now we will build up the options fro our select only adding one if the pirate doesn’t have the raid already.

We will need to pass all the raids into PirateDetail as props.

In PirateContainer add the prop to the route.

*// PirateContainer*

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** id **=** props.match.params.id;

**const** pirate **=** **this**.findPirateById(id);

**return** **<**PirateDetail pirate**=**{pirate}

onDelete**=**{**this**.handleDelete}

onUpdate**=**{**this**.handleUpdate}

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

}}/>

Ok so now we have the raids to populate the drop down.

*// PirateDetail*

**const** editUrl **=** "/pirates/" **+** **this**.props.pirate.id **+** "/edit"

**const** raidOptions **=** **this**.props.raids.map((raid, index) **=>** {

**if** (**!this**.pirateHasRaid(raid)){

**return** (

**<**option key**=**{index} value**=**{index}**>**{raid.location}**<**/option>

)

} **else** {

**return** **null**

}

})

Now we can add the drop down.

This will be added inside a form just after our pirates raid list.

*// PirateDetail*

**return** (

**<**div className **=** "component"**>**

*// AS BEFORE*

**<**/ul>

**<**form **>**

**<**select name**=**"raids" **>**

{raidOptions}

**<**/select>

**<**input type**=**"submit" value**=**"Add Raid"**/>**

**<**/form> /**/** ADDED

**<**button onClick**=**{**this**.deletePirate}**>**Delete {**this**.props.pirate.firstName}**<**/button>

*// AS BEFORE*

)

Now when a raid is added we can get it from the index and push to our pirates raids. Then call update. We will do this in a method called handleSubmit. Again bind this in constructor as well!

*// PirateDetail*

**constructor**(props){

**super**(props)

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

**this**.handleSubmit **=** **this**.handleSubmit.bind(**this**); *// ADDED*

}

pirateHasRaid(raid){

*// AS BEFORE*

}

handleSubmit(event){

event.preventDefault();

**const** index **=** parseInt(event.target.raids.value)

**const** raid **=** **this**.props.raids[index];

**this**.props.pirate.raids.push(raid)

**this**.props.onUpdate(**this**.props.pirate);

}

And add the submit event to our form

*// PirateDetail*

**<**form onSubmit**=**{**this**.handleSubmit}**>** *// AMENDED*

**<**select name**=**"raids" **>**

{raidOptions}

**<**/select>

**<**input type**=**"submit" value**=**"Add Raid"**/>**

**<**/form>

We should now be able to add the raids to the pirate as well.

**Summary**

* Learned how to add a raid to the pirate.

# React Requests - Add / Remove Raids

## Learning Objectives

* Understand how to add or remove raids to a pirate.

## Duration

60 minutes

# Intro

So our form works to edit a pirates basic details but we still want the ability to add / remove raids.

Instead of doing this in the form we will add this functionality to our pirate details view.

Next to each raid that pirate is on we will add a delete button.

And we will add a drop down of raids to add that pirate to.

# Delete Raids

Let’s start by deleting raids.

When we delete a raid from our pirate we will need to update that pirate in the database. So we will pass our handleUpdate method from the container to our pirate details.

*// PirateContainer*

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** id **=** props.match.params.id;

**const** pirate **=** **this**.findPirateById(id);

**return** **<**PirateDetail pirate**=**{pirate}

onDelete**=**{**this**.handleDelete}

onUpdate**=**{**this**.handleUpdate} *// ADDED*

/>

}}/>

Now in the PirateDetail we will add a delete button to our list of raids.

*// PirateDetail.js*

**const** raids **=** **this**.props.pirate.raids.map((raid, index) **=>** {

**return** **<**li key**=**{index}**>**

{raid.location} **<**button**>**Delete**<**/button>

**<**/li>

}) *// AMENDED*

Ok we can see this button and when we click nothing should happen yet.

When we click this button we will need to remove the matching raid in the pirates list of raids.

In order to do this we will need to pass the raid index to a method and use splice to remove from our pirates raid. Then call the update method we passed in as a prop.

We will write the method first. Remember to bind in the constructor!

*// PirateDetail*

**constructor**(props){

*// AS BEFORE*

**this**.deleteRaid **=** **this**.deleteRaid.bind(**this**)

}

handleDelete(){

**this**.props.onDelete(**this**.props.pirate.id)

}

deleteRaid(raidIndex) {

**this**.props.pirate.raids.splice(raidIndex, 1)

**this**.props.onUpdate(**this**.props.pirate)

}

Lastly set our button to trigger this method on click. In order to pass the index as a callback we can’t simply write something like

**<**button onClick**=**{**this**.deleteRaid(index)} **>**

As this would trigger write away.

What we **can** do is pass it into a callback for onClick. Like this

onClick**=**{() **=>** **this**.deleteRaid(index)}

Let’s add that to the button.

*// PirateDetail*

**const** raids **=** **this**.props.pirate.raids.map((raid, index) **=>** {

**return** **<**li key**=**{index}**>**

{raid.location} **<**button onClick**=**{() **=>** **this**.deleteRaid(index)}**>**Delete**<**/button> /**/** AMENDED

**<**/li>

})

Awesome We can now delete the raids attached to our pirate.

Let’s add the ability to give them some more.

## Adding Raids

We can add a drop down to our pirate to list the raids they don’t already have.

Let’s start by writing a method that checks if a pirate already has a raid. We can use the arrays some method to do this easily.

*// PirateDetail*

deleteRaid(raidIndex) {

*// AS BEFORE*

}

pirateHasRaid(raid){

**return** **this**.props.pirate.raids.some((pirateRaid) **=>** {

**return** raid.id **===** pirateRaid.id

})

}

And now we will build up the options fro our select only adding one if the pirate doesn’t have the raid already.

We will need to pass all the raids into PirateDetail as props.

In PirateContainer add the prop to the route.

*// PirateContainer*

**<**Route exact path**=**"/pirates/:id" render**=**{(props) **=>**{

**const** id **=** props.match.params.id;

**const** pirate **=** **this**.findPirateById(id);

**return** **<**PirateDetail pirate**=**{pirate}

onDelete**=**{**this**.handleDelete}

onUpdate**=**{**this**.handleUpdate}

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

}}/>

Ok so now we have the raids to populate the drop down.

*// PirateDetail*

**const** editUrl **=** "/pirates/" **+** **this**.props.pirate.id **+** "/edit"

**const** raidOptions **=** **this**.props.raids.map((raid, index) **=>** {

**if** (**!this**.pirateHasRaid(raid)){

**return** (

**<**option key**=**{index} value**=**{index}**>**{raid.location}**<**/option>

)

} **else** {

**return** **null**

}

})

Now we can add the drop down.

This will be added inside a form just after our pirates raid list.

*// PirateDetail*

**return** (

**<**div className **=** "component"**>**

*// AS BEFORE*

**<**/ul>

**<**form **>**

**<**select name**=**"raids" **>**

{raidOptions}

**<**/select>

**<**input type**=**"submit" value**=**"Add Raid"**/>**

**<**/form> /**/** ADDED

**<**button onClick**=**{**this**.deletePirate}**>**Delete {**this**.props.pirate.firstName}**<**/button>

*// AS BEFORE*

)

Now when a raid is added we can get it from the index and push to our pirates raids. Then call update. We will do this in a method called handleSubmit. Again bind this in constructor as well!

*// PirateDetail*

**constructor**(props){

**super**(props)

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

**this**.handleSubmit **=** **this**.handleSubmit.bind(**this**); *// ADDED*

}

pirateHasRaid(raid){

*// AS BEFORE*

}

handleSubmit(event){

event.preventDefault();

**const** index **=** parseInt(event.target.raids.value)

**const** raid **=** **this**.props.raids[index];

**this**.props.pirate.raids.push(raid)

**this**.props.onUpdate(**this**.props.pirate);

}

And add the submit event to our form

*// PirateDetail*

**<**form onSubmit**=**{**this**.handleSubmit}**>** *// AMENDED*

**<**select name**=**"raids" **>**

{raidOptions}

**<**/select>

**<**input type**=**"submit" value**=**"Add Raid"**/>**

**<**/form>

We should now be able to add the raids to the pirate as well.

# Summary

* Learned how to add a raid to the pirate