React Fundamentals

by Liam McLennan

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

My name is Liam McLennan and welcome to my course, React Fundamentals. I am the chief technology officer at StartsAt60. com I have a knack for seeing the potential in tools and techniques, having published Pluralsight's first courses on React and JavaScript. In this course, we are going to learn the basics of React and develop a simple web-based game. Some of the major topics that we will cover include: React components, JSX syntax, implementing forms, and application state management. By the end of this course, you'll know how to build a maintainable user interface with React. Before beginning the course, you should be familiar with JavaScript and web development. I hope you'll join me on this journey to learn React with the React Fundamentals course at Pluralsight.

Introducing React

Introduction

Hello and welcome to the first module of React Fundamentals. In these early days of client-side web interfaces, frameworks and libraries are evolving fast and free. Some large frameworks cover the entire client-side requirements such as AngularJS, em-ba, and Meteor. React has taken a different approach. It focuses narrowly on the rendering and event handling of client-side user interface components, leaving the program up with a responsibility and opportunity to choose other specialized components to complete their application. Which approach you prefer will depend on your style and the requirements of your project. The plan for this module is to introduce and lightly show off React. We will look briefly at the advantages and disadvantages of React. While React is a general purpose browser application UI library, it might not be suitable for every project or team. To understand the best way to construct React applications, we need to understand a little about how React applications work so we will examine the architecture of a React application. With a rapid pace of change in JavaScript libraries, it is impossible to provide an exhaustive comparison. However, it is useful to understand the high-level differences between major libraries so I will briefly contrast React with its major competitor, Angular.

What Is React?

React is an open-source JavaScript library for creating rich user interfaces that run in user's web browsers. It focuses solely on providing rendering and event handling functionality. Rendering is the conversion of data that describes the state of the user interface into document object model objects that the browser can use to produce a user interface that the user can see and interact with. This is similar to what view engines like Handlebars or ASP. NETs raise or do. React's event handling lets the programmer detect when a user interacts with their program and to specify how the program should respond. React was created by and is maintained by Facebook. It's a significant piece of Facebook's technology repertoire used in many of their projects. React appeared with many novel and revolutionary ideas that have helped to drive it's steady but sure growth in popularity. Its architecture borrows many ideas from functional programming including modeling components as functions, programming by transforming values, and separating the calculation of UI changes from the application of those changes. React popularized the counterintuitive notion of one-way data flow. Prior to React, two-way data binding had become a defacto standard. However, the designers of React realized that the productivity advantage of two-way data binding was not enough to justify the additional complexity that it adds to applications. By restricting applications to a one-way data flow, React can enforce a symmetry between the UI model and the rendered user interface. This property leads to React's greatest strength: the way in which it simplifies the programming of user interfaces. The Virtual DOM is the

JavaScript object model that React uses to calculate user interface changes. By performing this calculation logically, React can free the programmer from tracking individual UI changes and still be fast. React has no special UI template syntax. Instead, it relies on regular JavaScript. If a piece of UI is conditional, use a JavaScript conditional expression. To render a collection of things, use the JavaScript arrays map function and so on. If you've used Backbone, it's a good example of an imperative-style framework. You work with Backbone by specifying viral events that when one thing happens, then something else should happen in response. For example, when I select a row of a table, then a div should be updated. React chooses instead to have a declarative style. A React application is a set of components, each of which declaratively defines a mapping between some states and the desired user interface. The interface is only changed by changing the state. The declarative style has the advantage of being easier to reason about and easier to prevent bugs. Being easier to reason about means that the code makes it obvious how it works. It's easy to see how code changes or events will affect the program's outcome. The trade off is often that the declarative style requires the programmer to think through the problem more completely at the start. Once you understand how the program needs to work, you can represent that with React's declarative style. The result is a program that communicates how it works without requiring the reader to trace through code and track changes over time. React components are self-contained units of functionality. They publish a simple interface that defines their inputs as properties and their outputs as callbacks. By implementing the interface, React components can be freely nested within each other. With a bit of thought, composition can lead to a lot of easy reuse. Consider how much use Facebook get out of the like and comment components.

Demo: Setting up a React Development Environment

When it was first released in 2013, React was a simple JavaScript file. All that was required to use it was to add it to a page. Both React and the broader JavaScript ecosystem have become more sophisticated since then and it is no longer practical to use React without a build system. The responsibility of the build system is to process many JavaScript files written with modern JavaScript such as ES2016 and React's JSX syntax and convert them to a format that could be loaded into browsers. Several hundred open source projects exist to solve this problem for you but create-react-app, the React application bootstrapper released by Facebook has become the defacto standard. create-react-app is a quality build system but the application of bootstraps is extremely simple, lacking the functionality necessary for real-world applications such as state management and routing. To build an application on top of create-react-app, you will need to add and configure many other libraries yourself. Let's take a look at how to get started with create-

react-app. This is the GitHub project page for create-react-app. The ReadMe file explains how to install create-react-app with npm and to bootstrap a new project. So let's try that now. The first step is to install the create-react-app package from npm. So you'll need to already have Node. js and npm installed. We use the -g flag to indicate that we want create-react-app to be installed globally which means that the binary executable files will be available from anywhere. Now that we have create-react-app installed, we can use it to bootstrap a very simple React application. We will build a click counter that counts the number of times that a UI component is clicked. To create a new application, we use create-react-app followed by the name of our application and now, create-react-app will create a skeleton application for us in a new directory. Now that it's finally finished, we can change into the directory containing our application and start the application with npm start. As you can see, our application is up and running on localhost port 3000 and the instructions here say that to get started, I should edit the file, src/App. js so let's go have a look at doing that. I've opened the directory containing my code in visual studio code but of course, any text editor will do. The instructions told me to edit src/App. js and here, we see our first real React component. This is the one that's generated as part of the application skeleton and it's much too complicated for our purposes so I'm going to start by deleting the content of this component. I'd like my component to render a div so I'll create one of those and inside the div, I'll put some content. This div has been clicked some number of times and then we close the div. Don't worry about this exact syntax. We'll be covering it later in great detail. For now, I just want to demonstrate the process of creating a very simple React component and React application. Switching back to our running application, note that it's automatically reloaded and is showing the content that I've just created. I've mentioned a few times that a React component is responsible for converting a model of the UI into a user interface running in the browser. That model is typically represented as a JavaScript object so let's create a model for this application. Since all our application will do is track the number of times that a div has been clicked, the model can be quite simple. I'll call it model. It's an object. Has a single property called clicks which starts with the value zero. Now, when we render our React component, I want to pass that model data to the component. And now you can see that our application has changed slightly and it now says that this div has been clicked zero times. I wonder therefore, what would happen if I changed the state of this model to say, five, and now my application says this div has been clicked five times. So the running application is reflecting the model data that I set. So now the next step is that I would like to modify the application such that when I click on this div with my mouse, the counter increments. This of course involves event handling. I go to my app component and I'll set an event handler on the div. It's the click event I care about so I'll use the onClick handler and I'll blind that click event to a function called onClick. Now, I have to modify the usage of that

component to pass in a function called onClick. Now when the user clicks on div, I'd like to modify the model by incrementing the clicks property by one. That will work to update the model but there's still something missing and that is I have to tell React that when the model changes, it should refresh the interface. To do that, I put my render code inside of a render function and then immediately invoke that function. Then, within my event handler, I can also call that function. So this means that when the application starts, the user interface is rendered. When somebody clicks on the div, the model is updated and the user interface is rendered again. I will reset the model to the correct starting value and then jump over to the application. Let's try clicking on the div. When I clicked on it, it incremented the counter. Every time I click, it keeps on incrementing. So that's it, our very first and very simple React application.

Advantages and Disadvantages

By far, the greatest strength of React is its conceptual simplicity. Our React application is a tree of React components. A React component is a function that converts a model object into a piece of user interface, that's it. Everything else is window dressing. Speed is relative and React is optimized for simplicity, not for performance. However, React has always performed well relative to its competitors. React has a very clever algorithm for working out what changes it needs to make to the UI and minimizing work as much as possible. This results in a good default baseline of performance for applications that deal with very large document object models, there are ways in which the programmer can provide hints to help React perform even better. Because React is so conceptually simple, it provides some UI data to React components and get a bit of UI. It's easy to run React server-side. This is useful for optimizing the initial performance of an application and for making content available to search engine crawlers and other automated browsers. I refer to React as a library, not a framework because it makes no attempt to assemble a complete toolkit of everything you need to build a user interface. Out of the box, React is only suitable for very simple components. To build a moderately complex application, you will likely need to add a router, a system for managing state changes, validation, form support, and others. Of course, this means that you can assemble exactly the system you want. So it is both an advantage and a disadvantage. As I have already said, React is optimized for simplicity and correctness, not for developer productivity. If you want to smash out very simple applications, then React will not be the easiest way to do it. Dropping React into an application is non-trivial. You need to build process that can transpire old JavaScript versions and JSX and do the packaging into scripts that will work with browsers.

Architecture

A React application is a composed set of components. Each component has the design represented in this diagram. The inputs to a component are properties referred to as props and state. The difference between the two is that state can change. As much as possible, you should design components that don't use state because they're wonderfully simple. Note that the flow of data is all in the same direction. From the model via the render function to the document object model. Props and state collectively represent the model. The document object model or DOM is the direct result of rendering that model. For a given model, the rendered document object model will always be the same. The way to change the document object model is to change the model. Once the document object model is rendered, it can generate events which feed back into the component state and trigger another render cycle. Conceptually, we can imagine that for any state change, React will regenerate the entire document object model but if React actually worked that way, the performance would be poor. Instead, React maintains it's own document obstruction. A component's render function updates this fake document objects model known as the Virtual DOM which is extremely fast. Once that happens, it is the job of the React framework to compare it's fake document object model to the current state of the real document object model and update the real document object model in the most efficient way possible.

React vs. Angular

React and Angular are by far the two most popular JavaScript user interface libraries at the moment so I will briefly contrast them. Contrasting anything to Angular is difficult as Angular is a label applied to several completely different libraries as it is evolved from AngularJS, Angular 2, and finally, just Angular. As Angular has evolved, the gaps between React and Angular have shrunk as Angular has borrowed many successful ideas from React such as the preference for one-way data binding and component-based user interfaces instead of model view controller. React renders UI and handles events. Nothing else. You will need to choose many other libraries and integrate them if you wish to build a substantial application. Angular is a complete UI framework. Everything you need to build a JavaScript user interface is in the box, ready to go, integrated, and documented. React uses plain JavaScript for view logic. There are no restrictions. Angular uses something that looks like JavaScript but isn't. To quote the Angular documentation, "You write these template expressions "in a language that looks like JavaScript. " Many JavaScript expressions are legal template expressions but not all. In addition, Angular includes a vast array of custom template syntax. In addition, Angular includes a vast array of custom template syntax using asterisks, square brackets, round brackets, combinations of square and round brackets,

dollar signs, pound signs, and colons. React applications are written in JavaScript. Usually, a modern version that is then transpired down as part of the build process. Angular applications are written in TypeScripts, Microsoft's statically typed super set of JavaScript. This helps Angular developers to structure their applications logically around types and type transformations and also, to avoid a huge class of bugs. It is possible to develop React applications with TypeScript and Angular applications with JavaScript but these are the defaults of their communities.

Summary

React is a popular and mature open-source library for creating JavaScript user interfaces. In this module, we used the create-react-app project to create a React app including all of the build process, development infrastructure, and an application skeleton. We discussed the strengths and weaknesses of React along certain dimensions. We looked at the architecture of a React component and how you should work with that structure to get the best result. Finally, we contrasted React and Angular, two immensely popular libraries that have converged over their shared history.

Components

Introduction

A React Application or widget is built from a set of components. This module will introduce components and then create the foundation for the rest of the course. Every React Application has a single root component. Typically, that root component contains other components that in turn contain more components all organized into a tree. We will discuss what a component represents in a React Application. First, we must define the component then we can use the component by rendering that component into the DOM. We will discuss what a component's props are and what lifecycle events are available for our component. React components have a facility called State which we will look into before finishing with a brief look at some of the options available for testing components.

What Is a Component?

Components are the fundamental unit of a React Application. They are both simple and powerful. Each component corresponds to an element in the DOM. The component is responsible for rendering the content of that element and for handling any events that occur within it. Components can be nested inside other components. This is what is meant by composing components and it is a powerful technique for achieving reuse. Nested components correspond to nested DOM nodes. The outer component is said to own the inner component.

The Author Quiz

The Author Quiz is a sample app that we will build through this course. It's a simple game that displays the image of a famous author. The player has to select from a list the name of a book written by that author. In the example shown, we're looking at William Shakespeare so either Hamlet or Macbeth is the correct answer. If the player chooses an incorrect answer a red highlight is displayed. If the player chooses a correct answer a green highlight is displayed and the continue button appears. Clicking continue causes a fresh game to be loaded. Let's have a look at the completed application. This is the completed Author Quiz application that we will be working on throughout this course. When the page first loads it starts a game. This time it has randomly chosen a photo of Mark Twain. Did Mark Twain write The Shining? Apparently not but he did write The Adventures of Huckleberry Finn. A correct answer shows a green highlight and the continue button. If I choose continue I get a new game loaded.

Defining a Component

This is how we define a simple react component that renders a heading containing the message Hello at followed by what is presumably a timestamp. This syntax is initially confronting because it looks like a strange mixture of JavaScript and HTML. The value return from the function is actually JSX, a special markup language that React compiles to JavaScript. This is one reason why React requires a compilation step. Something must convert the JSX to JavaScript that can be understood by JavaScript interpreter. This is one of the two possible syntaxes for defining a React component. This syntax is nice because it is simple and because it clearly shows that a React component is a function from some model data to a piece of UI, here represented as JSX. Model data is passed into the component as the argument to the function. Curly braces are used to indicate a JavaScript expression. It should be evaluated and interpolated into the output. Here it is inserting the now property of the props object.

Rendering a Component

This is the complete code required to render a component. Here is the Hello component from the last slide. We need to import the ReactDOM module to get access to the render function. It is required to import the React module anywhere that JSX syntax is used. Remember that JSX is the markup language that looks like HTML but can be specified inside of JavaScript code. With that in place we can use the render function from the ReactDOM module to render a React component into a DOM element. The first argument to render is a JSX expression. This one says that we want to render the Hello component. Now is a value that we supply to the Hello component. Here it has a value that is a string representation of the current time. The second argument to the render function is a DOM element. The React component will be inserted into this DOM element. I've taken the code from the slides and set it up here in CodePen. At the top of the screen we have our JavaScript code and at the bottom of the screen we have a window showing the output. So you can see that our Hello component when it's rendered into the root DOM element it is displaying this text Hello at and then a timestamp showing when the page was loaded.

Bootstrapping the Author Quiz

Now that we have enough knowledge to be dangerous let's begin creating the Author Quiz application. First, we will use Create React App as shown in the last module to initialize and empty application skeleton. Next we will add the Bootstrap CSS framework to help set some basic styles and to provide support for responsive design. Finally, we will define our top-level AuthorQuiz component with some placeholder content. There is a lot more to follow before the application is complete but this is enough to get us started on our way and to build a strong starting point that actually runs. To begin our Author Quiz sample application we use create-react-app to create a new application called authorquiz. We can now follow the instructions and change into the newly created authorquiz directory and then start our application with npm start. And this will just verify that the installation was successful and everything is working properly. Okay, great. There it is, our new application up and running. Now the next thing I'd like to do is add Bootstrap for some basic CSS styling support. So at getbootstrap. com we can read all about the Bootstrap CSS framework. And on the homepage if we scroll down, in the installation section there's an option to use the Bootstrap CDN and it includes a URL to the Bootstrap CSS. So I've highlighted that URL, I'll copy it to the clipboard and go back to my terminal. I stop our application for the moment. Once I have that URL I can paste it into my browser. And then save that CSS file into the source directory of my application. Now that we have the Bootstrap CSS file

local in our application we need to include it into our application and the build process so that those styles apply. I've opened the Author Quiz application in Visual Studio Code. You can see the files that make up my application on the right hand side. By expanding these source directory I can find the source to App. js and this is the main component of our application. You can see already that it has an import for a CSS file, App. css. I'll leave that one there and I'll add a second import this time for the Bootstrap CSS file that we just downloaded. And now switching back to the running application I can see that the Bootstrap styles have been applied. If I open the developer tools in my browser, have a look in the style editor, the third one is the Bootstrap style sheet. So, confirming that it's definitely being loaded and used. Because we're building an Author Quiz application I'd like to rename some of these components. So, I'll rename App. js to AuthorQuiz. js. The corresponding test file becomes AuthorQuiz. test. js. And where the app component is used in index. js. I changed that reference to AuthorQuiz as well and also changed the import. Within the Author Quiz module I'll rename the app component to AuthorQuiz and make that the default export. And finally, I would like to remove the content of the AuthorQuiz component, replace it with a div containing the word Author Quiz. Let's jump over to the browser and make sure that everything is still working, and it is. Okay, so our app has reloaded and now all it contains is a div with the words Author Quiz. It's not much but it's a start that we can build on through the rest of this course.

Props

JSX can represent two types of elements. As in this example, React elements can represent DOM tags like this div. Elements that represent DOM tags are written in lower case. Attributes passed to these elements are set on rendered DOM nodes. This example will render a div with an ID attribute with the value mydiv as per the JSX. The other type of React element, user-defined elements such as Sum here, must have an identifier starting with a capital letter. Attributes on user-defined elements are passed to the component as a single object here given the identifier props. Props is an object containing two properties, a and b, with the values four and two respectively. Given those properties, the Sum component can render its output. Let's see the demo. Here we have the component from the slides called Sum. And when it's rendered we specify to props a which is four and b which is two. Four and two both as JavaScript numbers. Within the Sum component we have access to the props object and on that props object we can access the values a and b. So within our h1 tag we render a plus b is equal to the sum of a and b or four plus two is equal to six. The React documentation states all React components must act like pure functions with respect to their props. There are not a lot of rules but this one is clear. It

says that for a given props object the output should always be the same. This allows React to optimize component rendering. If the props have not changed then a component does not need to be re-rendered since its output should be the same.

Class Components

The preferred way to define a React component is using the function syntax we've seen so far, where a React component is a JavaScript function from a props object to a React element. It is also possible to define a React component as an ES6 JavaScript class. The class component syntax has some additional features that you will need occasionally. Here I have converted the Sum component from the function syntax to the class syntax. The minimal requirements for a class component that it must extend React. Component and must have a render method that returns a React element. Where possible you should use function components because they are more concise and they encourage better design.

Component Lifecycle

React class components allow you to override lifecycle methods if you need to perform operations at particular times in a component's lifecycle such as when it has just been created or immediately after it has been inserted into the DOM. Methods with will immediately before the verb are called immediately prior to that verb happening. For example, componentWillMount is called immediately prior to the component mounting. Methods with did immediately prior to the verb are called immediately after that verb happens. For example, componentDidMount. Component lifecycle methods are useful when you are implementing a React component to wrap an imperative API. For example, you may be creating a React component for a jQuery plugin. You would use the component lifecycle methods to initialize the jQuery plugin and possibly to remove it when it is no longer required. This slide shows some of the lifecycle hooks that are available during the mounting and updating phases of a component's lifecycle. Check the React documentation for the complete list.

State

In addition to props React components have another way of holding data called State. Props are values passed in by a component's parent. Its state is local, mutable data that can be created and modified within the component. Having local data that can change within a component increases complexity and limits the composability of the component. As such, you should avoid using state

when possible which is nearly always. In module one we saw a React component that counted the number of times it has been clicked. If we wanted that component to be self-contained we could move the click count into the component state. To use State we have to use a class component. In the class constructor we initialize the component state to a sensible default. In this case an object with a clicks property with the value zero. When we display the click count we read the value from the state property not from props because clicks is an internal state value, not an external props value. In the onClick handler we use the setState function to set the state to a new value. It's important to use the setState method rather than updating the state directly so that React knows that the state has changed and the component should be re-rendered. It's usually better to store and manage application state centrally so avoid class components and state whenever possible. This is the ClickCounter component implemented with local state. As we saw in the slides we have a constructor for the class which initializes the clicks to zero. When the component is rendered it outputs some text, this div has been clicked, followed by the clicks value from the local state. I test it out, say each time that I click on the component the number of clicks is incremented. And that's happening via the onClick handler on this div and every time there's a click event we handle that event, we call setState and we set the state to whatever the clicks value was before plus one.

setState

The setState method merges the new state with the old state. All of the previous state remains unless it is overwritten. Consider a state object with properties a and b having values one and two respectively. Calling setState with the properties b equals three and c equals four produces a new state object. The property a is unchanged. The property b is overwritten by the new state and property c is newly added. In addition to changing the state of the component setState also causes the component to be re-rendered eventually. For performance reasons setState calls a batched. There is no guarantee that the state change will occur immediately.

Prop Validation

Recall the Sum component we defined earlier that adds two numbers and displays the result. For this example because the prop values supplied are four and two, the output will be four plus two is equal to six. If we change the values of a and b to key and board the component still works. The output will now be key plus board is equal to keyboard. Now, this may be the desired behavior but we often want to be more precise in our use of types. This just happens to work because the

plus operator is defined for both numbers and strings. This is the code from the slides. Note that a prop is set to the string key and the prop b is set to the string board. The Sum component pluses a and b together and we get the output, key plus board is equal to keyboard. If we make a, the letter a and b the number two, now we are adding the letter a and the number two. At best this is ambiguous, at worst it's nonsensical. The output of this example is a plus two is equal to a2. This result occurs because when confronted with a string and a number to add together JavaScript converts the number to its string representation and concatenates the string. But this example is very likely an error. Our Sum component is meant for adding numbers. So how do we constrain a and b to being numbers? React components can validate the props they are passed by defining a PropTypes object. PropTypes specifies the allowed values for each prop key. Note that this is runtime validation. If the component is passed invalid props then PropType validation can provide a helpful error message but it won't prevent the error from occurring. React provides many validators. You can validate the data type of a prop that a prop is supplied, that a prop is an instance of a particular class or you can provide a custom validation function. Additional benefits of specifying PropTypes include making the component's implicit dependency on props explicit and documenting the expected use of the component. Note that PropTypes is defined in a separate npm package called prop-types. To use it you must install the package with npm install prop-types. Then import PropTypes from that package. Here is our Sum component upgraded to include PropType validation. The definition of the component hasn't changed at all but once we have the sum function we then add a PropTypes property to the function. The value of PropTypes is an object containing our validators and they're specified with a key matching the prop name and a value specifying the required validation which come from the PropTypes package. For our Sum component I'm using the PropTypes. number validator to indicate that both a and b should be numbers and I'm further restricting the value of a and b by saying that both of those props are required. In my usage of Sum I give a the value c and b the value two and you can see in the console that we get a runtime warning. Failed prop type. Invalid prop a of type string supplied to Sum, expected number. So that's quite a useful error message. It's pointing me to the prop a as being the one with the error and saying that it expected a number but instead got a string. If I correct that the error goes away. TypeScript from Microsoft and Flow from Facebook are related efforts to add static type checking to JavaScript. They provide an alternative way to verify that a React component is being used correctly. In a way, they are superior to specifying PropTypes because they allow problems to be detected at compile time while the developer is writing the code. This example is written in TypeScript. This syntax indicates that props is of the type SumProps. SumProps is defined above as an object having two number properties, a and b. Because we are creating a Sum element with an a value that is a string when it should be a

number TypeScript provides this compile time error message. Effectively saying that a should be a number but is being supplied as a string. This is a React course not a TypeScript or Flow course so I will not cover statically typed JavaScript in any more depth. Suffice to say that TypeScript and Flow are excellent additions to a React project. They provide sophisticated type system that help you to model your domain and reason about solutions, as well as preventing lots of type-related bugs.

Testing Components

Testing should focus on application logic. We test that our applications are correct according to their requirements. React is a user interface technology. An application's logic should not be implemented in its user interface, and therefore, React components should generally not contain application logic and are not a high value testing target. Having said that, it can be useful to test React components and there are properties of React components that make them amenable to testing. In particular, the function components that we favor are a simple function from props to JSX. They're easy to test by rendering the function with a set of props and making assertions about the result. We can also simulate events like the user clicking a button and make assertions about what happens. Let's try adding test to one of our example components. I have added a file called Hello. test. js to a Create React App application. Because this is a Create React App application it's already setup to be able to run test. Create React App uses Jest as the testing framework which is another Facebook project. If you search the web for Jest you'll be able to find this website which explains how to get set up with Jest and documents, how to write test with Jest, and including all of the different types of assertions that you can use. But as I said, Jest is already setup with Create React App. To be able to demonstrate how the testing framework works I've created one simple test that I expect to fail. You can read more about Jest but at a very high level tests are written using this it function where we make some kind of assertion about what we're expecting, and then implement that in code. If you like you can group those tests by wrapping them in a describe function call. Describe is where I say I'm going to have a set of tests that are about setting up testing. And then the particular test that I've created is that it should fail if I expect one plus one to be three. I can run this test using npm test. And you can see that I have one test failed, one out of one because it expected three to be two. I always like to set up my test framework by testing a failing test case because that shows me that the system's working. If I was to start with a test that I expect to pass, then when the test run everything passes and I get less information about whether or not things are actually set up properly. So now let's make this more interesting and see how we can test a React component. Earlier we used a component called

Hello. It's a simple React function component that renders some content including a passed in timestamp. Let's begin our testing efforts with the simplest possible setup which I'll call when testing directly. And by this I mean that we're going to test our Hello component simply as a function without any particular helpers for React. Because it is a React component we do need to import the React library so that the React elements will be in scope. We have to write a number of tests and before each of the runs I want to run a setup function so I used the beforeAll helper, and within this helper I want to call my Hello component and pass in a props object. And the props object contains a timestamp as an ISO string. Now I can create my first test. I'd like to start with something simple so we will test that the Hello component returns a value. I make an assertion on the result and I'm simply going to say that I expect the result not to be null, does the result have a value? Okay and that test is passing. Now to get slightly more sophisticated we can assert that the result returned from the Hello component is a h1 element. To do that I look at the type of the element that's returned. I expect it to be a h1. Every time I save the tests are rerun and now I have two passing tests when testing directly return a value and is a h1, both passing. Moving along, I can test that element has children. To do that I look at the props of the result and the children property. And I say that I expect that to be truthy. So those are some simple tests that I can run just by testing my React function component as a function. Next I'll show how we can test a component with ReactDOM. And ReactDOM you'll remember is the module that we use to render a React component into a webpage. And I would like to test that it renders without crashing. To do this I need to create a DOM element which I can do using document. createElement. I'll create a div then I can use ReactDOM to render a Hello element and I render it into the div that I just created. When I did that I got an error. When I look at the detail of the error I can see that it's failing because ReactDOM is not defined which means that I've forgotten to import a module. If I add the missing import then all of my tests are passing. To do more sophisticated testing of React components there's a helpful library created by Airbnb called Enzyme. So the next thing I'd like to do is add Enzyme. To add Enzyme to my project I install the Enzyme module as well as the Enzyme adapter for React 16 because that's the version of React that I'm using. Now that that's done I can add another set of tests. When testing with Enzyme we will use a helper called shallow from the Enzyme module which is used to perform a shallow rendering of a React component. Once again, I'd like to test that the Hello component renders a h1 element. So, using the shallow function can pass in a React element to perform a shallow render and the result return from shallow is an object containing a number of helpful methods that I can use. In this case I'm using the find method to search for a h1. And I expect the length of the collection produced to be one. Once again my test have failed because I've forgotten to do something. When I look at the error message I'm told that Enzyme expects an adapter to be

configured. You might remember that we did install an adapter but then I've forgotten to configure it in the code. Fortunately this is easily fixed. Before I run my test I simply have to call the Enzyme. configure method and configure the adapter and that adapter also needs to be imported. And finally, I need to include the default export from Enzyme as well and now my tests are running. Finally, one last test I can use Enzyme to test that the content of my component is what I expect to be. So I would like to test that the result contains the string Hello at a particular timestamp. Once again, I shallow render the Hello element and then make an assertion that the result that I rendered contains an h1 element containing the text Hello at that particular timestamp, and that should be true. So that's just a quick introduction of some of the techniques that you can use to test React components. Now that we know how to test React components let's add some tests to our Author Quiz application. As I mentioned before, any file containing something. test. js will be detected by Jest and run as a collection of tests. So I've added a file, AuthorQuiz. test. js which we will use to test the AuthorQuiz component. So I start by introducing this test library as a collection of tests for the Author Quiz. As before, we'll start with an easy one and just test that our component can render without crashing. We'll use ReactDOM and the render method to render an instance or element of the AuthorQuiz component and that test runs successfully. Jumping over to the AuthorQuiz component now. We can see that it doesn't do very much so there's not a lot for us to actually test yet.

Summary

React applications are constructed as a hierarchical graph of connected components. Components can be defined using the function syntax or the class syntax. Prefer the function syntax. It is neater, more concise, simpler and helps to discourage bad practices. Components can render based on data from two sources, props and state. Props contain immutable data passed from parent components. State contains local mutable data. Avoid using state where possible. Props can be validated at runtime using the PropTypes library or they can be validated at compile time using a static type checker like TypeScript or Flow.

JSX

Introduction

JSX is an external domain specific language that is optimized for generating XML-like documents. That may not sound very exciting but keep in mind that our web application markup language is HTML, and HTML is XML-like. For a react application, JSX will typically be used to generate HTML. But it also supports custom react components and SVG. This module will describe JSX, provide strategies for using JSX, discuss attribute and child expressions, and color peculiarities of JSX such as illegal attribute names, unescaping contents and the style attribute. To develop a proper understanding of JSX, we will begin by investigating what JSX is and how it compiles to JavaScript. We will look into how to use JSX or not use JSX. How to pass data into components and out of components. And how to compose components into larger components.

What Is JSX?

JSX allows us to include XML-like syntax in line in JavaScript. React uses JSX to describe the composition of react components in a readable way. The JSX transformer preprocesses the JSX and converts each element into a JavaScript function call. Most projects use Babel to convert JSX to JavaScript. The typescript compiler is also capable of compiling JSX. So if you use typescript, you don't have to use Babel as well. JSX is a syntax included in line in JavaScript files that is converted by a preprocessor like Babel into regular JavaScript. Sum here is a react component. When a component is combined with some props it is called a react element. So this is what a react element looks like in JSX. Here is the equivalent react element in the compiled JavaScript. Each JSX react element is converted into a call to react. create element. The react component is the first parameter of the create element method. The props are collected into a JavaScript object passed to the second argument of the create element method. If the sum element had any children, they would be passed as the third parameter. The sum element does not have any children, so the third parameter is null. This example has the sum element wrapped in a h1 element. When this JSX is compiled to JavaScript, the value passed to the first parameter of create element is a string H1. Built in components, those that map to HTML elements are represented as strings. The H1 element has no props, so the second parameter gets a null value. The JavaScript representation of the sum element is passed as the third parameter because it is a child component of the H1 element. Even with this trivial example, we begin to see the value of JSX. The JSX version is easier to read and understand and the way that components are nested is communicated more easily. Here is an example we have already seen. This is our sum components which has two props A and B, and which outputs the sum of those two elements. The sum component itself contains a H1 and then it renders its result inside that H1. Here we see the same components in its compiled form. All of the JSX has been removed and replaced with pure

JavaScript. The sum component now returns a call to react. create elements. The first argument is the string H1, it's a string because H1 is a built in element. There are no props to that H1 and then the third argument is the children. Where we render the sum component inside of a div the first called create elements is to create that div. There are no props and then the third element is the children of the div in this case this is where we create our sum elements.

Not Using JSX

Not everyone likes JSX. Some people choose to use react but choose not to use JSX. For example, if you wanted to avoid a compilation step. We've seen how the JSX transformer converts JSX syntax into JavaScript function calls. If we want, we can simply skip directly to the JavaScript version. The typical workflow is to write components with JSX, process the JSX through the JSX transformer and finish with nested function calls representing react components. The alternative, is to bypass JSX completely and skip directly to the JavaScript function calls. JSX can be easier to read but there is a price to pay. The foremost problem is that a script that includes JSX is not valid JavaScript. It can't be minified, processed through a linter, debugged in the browser or formatted with a JavaScript syntax highlighter.

Props in JSX

To supply props to a react component, we specify attributes on the JSX elements. This is a JSX expression. Hello is the react component name. Now is the JSX attribute that will populate the now prop on the react component. The value assigned to the now attribute is the JSX attribute expression. Note that it is delimited by curly braces and not quotes. The expression inside the curly braces can be any valid JavaScript expression. When the component is rendered, the JavaScript expression will be evaluated and supplied as the value for the now prop on the hello component. JSX attributes can also be given a literal string value by using quotes instead of curly braces.

Spread Attributes

ES 2015 introduced a feature called Spread syntax that allows arrays and objects to be expanded in place by prefixing the value with three dots. With spread syntax, an array or object can be used when multiple values are expected. JSX supports spread syntax, so that a number of props can be specified by using the spread syntax on a single object. The sum component expects two props, A and B instead of specifying A and B separately we can use the spread syntax on an

object containing properties A and B. The two forms are equivalent. Sometimes the spread attribute JSX syntax is more convenient, especially when there are a large number of props involved. Here we have our familiar sum component and it remains unchanged. The difference in this example is that I've introduced a new props object and set my values A and B on that object. Then when I create the sum element, I use the spread syntax to expand the props object and supply the expected props A and B.

Events

So far, we have seen props used as a way to pass data into a component. Props are also the preferred way to pass data back out of the component. This usually happens in response to an event. To use props to pass data out of the component, specify a function as the prop and call that function. Pass the data as the function arguments, the clicker component has a single prop handleClick, which is a function. It renders three buttons, A, B and C. Each button has a click handler setup to call the handleClick prop. When the application is rendered, the handleClick prop is defined as a function that receives a letter and logs that letter to the console. So when I click A, the output is A clicked, B B clicked, C C clicked. So this is an example of the way in which we can pass a function in as a prop and then call that function as a way of getting data back out of the component.

React Data Flow

As we know, react applications consist of react components arranged in a hierarchy or directed acyclic graph. Data is passed down the component hierarchy by passing values into components props, data is passed back up the component hierarchy by passing values as function arguments to functions passed in this props. ClickyButtons is a react component with two props. Number of buttons is a value passed into the component that tells it how many buttons to render. OnSelection is a function passed into the component that is called when a button is selected and that is our mechanism for passing data back out of the components. The ClickyButtons component renders a div. Inside the div, I create a range from one to number the of buttons plus one and for each of those numbers, I call the make button function. And the make button function creates a react element when that element is clicked, the on selection prop is called and the ID of the button is passed as the function argument. So when I click the button 39, 39 is passed to on selection which is console. log and so 39 is logged to the console.

JSX and HTML

The panel on the left shows a label element expressed as JSX. The panel on the right shows the corresponding HTML generated by that label element. Attributes set on JSX dom elements, those are the built-in components with lowercase names that represent dom notes are passed through to the rendered HTML with a few important differences. Attribute names are written with camel case that is they start with a lowercase letter and each new word begins with an uppercase letter. Because for and class are JavaScript reserved words, they are not used as JSX attribute names. The for HTML attribute is replaced by the HTML for JSX attribute. The class HTML attribute is replaced by the class name JSX attribute. The style attribute is available but works differently. In HTML, this style attribute expects a CSS string as the value. In JSX, the style attribute expects a json object. With each key being a camel case CSS property and each value being a string CSS value. Note the double curly braces, this is because the outer curly braces delimited the JSX expression as usual. And the inner curly braces delimit the JavaScript object literal.

Unescaping Content

One of the nice properties of react is that it escapes all content by default, making it harder to fall victim to cross site scripting attacks. But sometimes you weigh the risks and decide that you have a valid reason to include unescaped content. Usually because you want to show somebody some user created content. Unescaped content means content that is not encoded to prevent cross site scripting attacks. Cross site scripting attacks are a major security vulnerability which is why the attribute for directly setting HTML from code is called dangerously set inner HTML, and it requires an object with an underscore underscore HTML property. You should avoid using this unless absolutely necessary. This is an example that shows the default behavior of react. The dangerous container component populates the content of a paragraph tag with a string of HTML. Because of react's default behavior to encode output, what we end up seeing is the actual HTML. This is safe, but may not be what we want. For example, what if I want the content to show as the word hello bolded? In that case, I can use the dangerously set inner HTML property. Instead of the usual react syntax, I provide an object with an underscore underscore HTML property, set to the content and I provide that to the dangerously set inner HTML property. This tells react that that content should not be escaped and in turn that means that the output is correctly formatted.

Child Expressions and Elements

JSX is hierarchical. So any valid JSX expression may be a child of any other JSX expression. You can have many JSX expressions or elements within a single JSX component. That is a JSX component may have zero, one or more direct children. This is necessary since JSX needs to be able to represent HTML. Without multiple child elements there would be no way to represent multiple items within a list or multiple rows in a table. Within a component, you can reference its child elements via the special property children. This means we can create components designed to wrap other components. For example, we might define a component that controls the visibility of its children. It doesn't matter what the children are, anything with in this component can be hidden and shown. Conditional display is a react component that shows or hides whatever child components are passed to it. It expects a prop value is visible that is a Boolean and is required. Conditional display displays its children if the isVisible prop has the value true. When we use the conditional display component, the child elements can be anything we want including nothing, one component, more than one component and components with their own nested children. Here we are providing two child elements to the conditional display element. When the isVisible prop is true, these elements will be shown. When is visible is false, these elements will not be shown. This example includes the sum component that we have seen a number of times before. And the conditional display component that we saw in the slides. When the react application is rendered, isVisible is set to the value of the show sum property of the state object. The children of the conditional display element are a H1 heading tag and a sum element. Finally, I have defined an interval and a callback that fires every two seconds, toggles the U value of show sum. So it'll be true for two seconds then falls for two seconds, then true for two seconds and so on. And also calls the render function to re-render the application. And as you can see, that has the effect of displaying the child elements for two seconds and hiding them for two seconds and toggling back and forth.

The Author Quiz

Now that we have covered JSX, let's revisit the author quiz application and upgrade it to render properly. When we last left it, the author quiz application was rendering a simple title. In this clip, we will add a header with a title and a brief description of how to play. We will add the author's photos, we'll add data including linking books and authors and we'll implement random game selection. In short, we will complete the noninteractive parts of the application. Welcome back to the author quiz application. This is how we left the application last time. we have our main author quiz component that simply renders a div containing the text author quiz. I'll begin by starting the application and that's how our application renders at the moment. It's just the text of the quiz.

The purpose of that was to get an application up and running with all the build tooling. But now that we've completed that step, I'll delete the content we had there and put in something a bit more meaningful. I'd like my application to be wrapped in a div with the container fluid class. This is a bootstrap class that specifies that I'd like a fluid layout for my application. Within that outer container, I'll include a few high-level structures. We'll have a hero component for our header, we'll have a turn component for the central game mechanics and we'll have a continue component for the button that moves the user along to the next game. When I save the file, I get some errors telling me that at lines nine, 10 and 11 you have errors. Hero is not defined, turn in not defined and continue is not defined. And that's because these are three react components that I'm yet to create. So we can go ahead and do that. Start with the hero component, the hero will be a bootstrap row, and within the row, I'll embed another div featuring some bootstrap classes. This is not a bootstrap course, but just briefly jumbotron is a class that applies for styling, col-10 means that the hero should be 10 columns wide out of a possible 12. And offset one means that it should be offset from the left by one column. Within that we'll have a H1 with our application title, and a paragraph of text explaining how to play the game. Select the book written by the author shown. And now I'm down to two errors. If I create a turn component and a continue component, now we see that the application builds successfully. The build succeeded but the application fails at runtime. It tells me that the turn component returns nothing from its render function. This is quite correct, okay. So to get the application to run, I will actually have to put in return statements with empty div. And the same for the continue component. There we go, now we have an author quiz with a styled header. I'd like to add one more component to the author quiz and that is a footer component. Once again, we create a function to represent the footer component. This is also a bootstrap row. And the row contains some content specifying the source of the images used. There we go, now the application has a footer as well. Next we will turn our attention to the turn component. No pun intended. And this is where the main activity of the game takes place. It includes the photo of a randomly selected author as well as a number of options of books at least one of which was written by that author. So instead of just rendering an empty div, we'll render a slightly more complicated div. This also is a bootstrap row and I'll apply a class called turn there as well, which we'll use to add some styling later. I also want to set the background color of the whole turn area, and ultimately that will be based on the results of the game. For now I'll just set it to white. Within the row, I'll add a column. Again using bootstrap, we'll make this column four of the 12 columns. So a third of the screen and offset by one. And within that left column, we'll put the author image. And the URL to the author image is something that will come from the props passed to the component. So I'll expect an author prop with an image URL property. Also want to add a class name there, again, so we can add some styling, author image. And alt text author.

That first column has already used up five columns out of the total 12, and now we'll create a second column, so it can be six columns wide. And in this right hand column, we'll take a collection of books again that we expect it passed in as props and we want to map over that collection of books and print out the titles in paragraph tags. The turn component as implemented is starting to show that there is some data that we need supplied as props. We need an author property, and we need a books property. So I'll add a props parameter to the turn component including author and the books. In the app it currently fails. Saying author is undefined and that's because when the author quiz is rendered, nothing supplies these props to the turn component. So let's fix that. We'll go all the way back to index JS, which is where our application is actually rendered. And this is where we'll define the application data. I'll call the value authors, and this will be an array of authors. There's some data for the first author. So we have the author's name, we have a URL to an image of that author, we have the source specified and we have a collection of book titles written by that author. Looking on the file system, if we open the public directory I've created a directory called images and inside that a directory called authors with a number of author photos including this photo of Mark Twain, which is the one referenced in the data. Now recall that the turn component expects an author object and a collection of books. So I'll create an object called state as in application state, which can be an object with those properties and author, I'll use our Mark Twain example. And books. And for this I'll just use Mark Twain's books. And on second thoughts, I'll move that into a property called turn data. I'll pass that state object into the author quiz component. This is using the spread operator syntax. To expand the state object out into its properties. So the prop received by the author quiz component will actually be turn data. Let's jump to the author quiz component now. Note that author quiz is implemented as a JavaScript class and that's not necessary. So I'll take a moment to convert that to a react function component. And as we just discussed, the prop where we called turn data we don't need any of that inheritance. We don't need the render method. And now I can pass the turn data value through to the turn component. Again using the spread operator which means that the props received by turn will be author and books. And now we have our application rendering with the author's photo and a list of book titles. In this case there's only one book in the list, styling is a little bit out, but that's easily fixed. I'll open the CSS file for the application and add some simple style rules. Then you can see the application is starting to look a little bit better. But the list of books is rendered as a list of paragraphs. And that's not very sophisticated. I'd like to create a book component which will have a title prop set to the title of the book. It needs to have a key prop because this is required by react. Any time that you render a collection of components, you have to provide a key prop with a unique value so that react can tell the different elements apart. I'll use the title of the book as the key because I know that that

will be unique, the build fails because book is not defined. So we can define book. Once again it's a function to represent a react component. The prop will be titled. The component returns a div with the class on it and inside the div, the heading containing the books title. The application still works, the build is successful. And with a little bit more CSS rules specified, the application starts to look a little bit more like the intended outcome. I would like to see this list of books with a few more items in it. So if we go back to index JS, where the application data is specified, all I need to do is add something to the list of books, added two more books for total of three. And so now they appear in the application. Now we have almost achieved our goal which was to complete the read side of the application. The application looks like it's rendering because it's got the styling, it's got the author's image and it has a list of books. But the problem is that the author isn't randomly chosen and the list of books are all written by that author. So the game wouldn't be any fun as it stands, since all of the answers are correct. What we'll add next is the random selection of the author, and also the construction of the set of possible answers. The first thing we need is a lot more author data. So I'll replace what we have with a larger set. So Mark Twain is still there, but now we have Joseph Conrad, JK Rowling, Stephen King, Charles Dickens and William Shakespeare. Each of whom have a set of books to find. Now instead of hard coding turn data to be the first author in the list. We'll use a function to choose the data for the game, which I'll call get turn data. So there I've used a function that doesn't exist which breaks the build and tells me that I should go ahead and write that function. Get turn data. What does this function need to do? Well the first thing it needs to do is select a set of possible answers. And we'll do that by joining together the lists of books written by all of the authors in our data set, shuffling them into a random order and then choosing the first four. I can build the collection of all books in our data set by reducing the authors collection and concatenating each author's books into the larger set. The next thing I want to do is shuffle that list into a random order, and that's a little bit more tricky. So I'll get a little bit of help from an external library by using npm install to install the underscore library. Now at the top of my script, I can import the shuffle function from underscore. The value for random books is equal to shuffle all books and then use the slice method to take the first four elements of the list. Now I'll choose a correct answer from the for random books. Use the sample function to choose a random value. Sample is also an underscore function. So I'll add it to the import. Finally, we're nearly ready to return the turn data required by the turn component which is your recall was a books value and an author value. Books is easy, you now four random books that we chose. And next we need the author. The way that we'll get the author is by using the find method of the authors array. To find an author such that the author's books collection contains a book where the title is equal to the answer that we chose. So this is essentially choosing the author that has written a book with the same name as the answer that we selected.

And that's how we find our author. So you can see that the output of the application's changed. We're no longer seeing Mark Twain, this is Charles Dickens. Now we've got four books, not all written by the same author. But where one of them was definitely written by Charles Dickens. If I refresh the browser, I get a different author image, JK Rowling. A different set of books and again at least one of them is a correct answer. I can see that the formatting here is a bit off. So I'll go ahead and fix that. That's better. The goal of this demo was to get the author quiz rendering a game that includes a little bit of styling, to show the name of the game and an instruction for how to play it. But also choosing an author, choosing a set of possible answers, obviously including a correct one and rendering the whole thing. So that is now complete.

Summary

JSX is a compile to JavaScript syntax used in react applications to describe react elements and how they are composed. This is possible to do directly in JavaScript without JSX, but JSX makes the result far more readable. Components are custom or built in dom components. Ultimately all components are composed of built in dom components. Props are used to pass data into and out of components. JSX elements can take other JSX elements as children forming a hierarchical tree structure.

Events

Introduction

Could you create a React application that makes no use of events? Sure! You could have a read only application. In this scenario react would function as an over complicated view templating engine. Comparable to Handlebars EJS, Razor ERB or Hammel. React is at its best in highly interactive user interfaces and that means dealing with events. Most developers will have worked with events before. However for completeness we will briefly discuss what events are and why we should care. The events generated by the browser DOM we will call DOM events. This includes things like button clicks, scrolling and change events. When we create our own components we can give them their own events. We will call these component events.

Events

In software development, events are an extraction representing something happening at a particular moment in time. An event is a value carrying data describing itself. An event could be a user action like a mouse click or a character types. It could be a piece of data arriving from the external system or a timing event. Event-driven programming is a programming paradigm in which the flow of control is driven by responding to events. User interface programming has a tendency to rely heavily on events since the program is interacting with a human and needs to respond to the users actions. Typically, the programmer registers functions to be called when a particular event occurs. Those call back functions then respond appropriately.

DOM Events

DOM Events are the events generated by the browser. Things like button click events, change events for text inputs and form submission events. All browsers provide an event based programming model. They all provide a similar set of events that behave in a similar way. So that the programmer does not have to understand and account for the specific quirks for each browser, React provides a normalized event abstraction called SyntheticEvents. DOM Events are the events that receive a react SyntheticEvents object. Events is a simple react component that renders a button. An event handler caller click handler is registered to be called when the buttons click events fires. Click handler receives a SyntheticEvents parameter and logs it to the console. Note the click handler is a function that takes a parameter and sends it to the console. Console. log is also a function that takes a parameter and sends it to the console. So we can simplify this code. Anytime you have a function that just forwards parameters to another function you can replace the outer function with the inner function. Here we have the events component from the slides. The events component contains a button and an event handler is registered for the onClick DOM event. When the onClick event occurs, click handler will be called. It will be passed a SyntheticEvent object and a SyntheticEvents objects will be logged to the console. Let's try it. When I click the button the event object is logged to the console. So that's an example of a reactive function component that handles a DOM event.

preventDefault

When an event occurs it may be desirable to handle the event and prevent the browser from doing what it would normally do. The SyntheticEvent object has a prevent default method for this purpose. The most common example of this is preventing forms from submitting. The default behavior of a form is to submit the page and reload. In a single page application this is usually

undesirable so we handle the form submission, cancel the browsers default behavior using the preventDefault method and submit the data as an A jacked request that does not result in the page reloading. Another example as shown here is the check box input. When clicked the browsers default behavior is the check the check box. This codes calls preventDefault which will stop the check box from being checked when it is clicked. This is the nocheckbox react component. I called it nocheckbox because unlike a check box this one can't be checked. Watch what happens when I click the check box. As you can see I click the check box but the check box is not checked. The reason why is because of what happens in the event handler we've registered for the onClick DOM event. Here is that event handler and what it does is call the preventDefault method on the SyntheticEvents objects. That cancels the browsers default behavior of checking the check box. Let's have a look at another example. This one is a Java script class style react component called reloader. Let's start by looking at the render method and we can see that the reloaded component brings a form. The form has an event handler for the on submit DOM event. It also has an event handler for the on changer events of its text input which is bound to the content property of the components state. And finally there's a submit button for submitting the form. The onChar event handler that gets called each time the user types a character into the text box, updates the component state with the current content of the input. The event handler that's registered for the form submission checks what the state of the content currently is and if it's not equal to reload then it cancels the default behavior for this event. Now the default behavior for form submission is that the page will reload. In effect this means that we have a text box that we can type into. When we submit the form, under normal circumstances the default behavior should be prevented and nothing should happen, but if the content of the text box is reload, then the default behavior won't be granted and the browser will reload. Let's try that. I'll submit the form and you can see that nothing happens. The page is not reloaded because the default behavior is prevented. This time I set the value of the value of the input to string reload. When I click the button, the page is reloaded.

Component Events

In addition to the browser events published by the react DOM components, components can publish domain specific component events. These events are typically at a higher abstraction than the browser events. Instead of events like button clicked, component events are likely to be things like task added or priority changed. In other words they express concepts from your application into main instead of raw events from the browser. In practice a component event is implemented by passing a function as a prop into a component and then calling that function when you want

to trigger the event. Component events are the fundamental technique of passing data out of a component. For example, you may have a component that includes a form and the component may publish a component event when the form is submitted that includes the form data. Let's say that we want a react component that omits an event when it's clicked an even number of times. We will create a component called EvenCounter. The EvenClick event is a component event. As promised here is the EvenCounter component. Once again I'm using a Java Script class style component and the reason is that I want the component to have local state. The local state for this component has one property called clicks which initially has the value zero. This property is used to count the number of times the component has been clicked. The component renders a div with an event handler I registered for the onClick event. Inside the div we render a string explaining the number of times the div has been clicked. The function that handles the click calculates a new value for the number of times the component has been clicked. It updates the state to that value then it performs a check to see if the number of times the component has been clicked is divisible by two using the modular operator. If the number of clicks is an even number then we call the on EvenClick prop which is a function and we pass to that function the total number of times the component has been clicked. Where we create an element fourth component we must provide a handler for the on EvenClick event, and within that handler we take the number of clicks and we write a log to the console. Even followed by the number of clicks. For this demo I've got the browser tools to the right so you can see a little bit more easily now. If I click the even counter components note that the value the number of times it has been clicked increments each time I click. Now if you switch your focus over to the browser console each time I click if the total number of clicks is odd then nothing is written into the console. If the total number of clicks is even, then that value is written to the console. So this component is demonstrating both a DOM event in the onClick event of the div as well as a component event in the on EvenClick event of our EvenCounter component.

The Author Quiz

With our understanding of DOM events and component events, we can now make the Author Quiz game interactive. When the player selects an answer we want the user interface to indicate if they were correct or not. To do this we will check the answer the player selected for correctness. If it is correct then we will change the game background color to green. If it is incorrect then we will change the game background color to red. I've opened the Author Quiz application in Visual Studio Code and this is the AuthorQuiz. js file. We're looking at the turn components which is the main component that renders the body of our game. In its outer div, we've got a style where we

set the background color to white. So this is where I'd like to be able to change the background color to green or red depending upon whether the user selected a correct answer or an incorrect answer. So I need to know if we're in a state where the user has selected a correct answer, the under has selected an incorrect answer, or the user hasn't selected any answer at all. And to do that let's add another prop into our components and we'll call this prop highlight and highlight will be a string that can take one of three values. None, correct or wrong. But now the problem is that we need to get to a background color and we're starting from a value indicating if the user has got the correct answer or not. So there's a mapping that needs to occur and I'll create a function to do that. The function, we'll call it highlight to background color because that's the mapping that I'd like to perform. To make this mass into claritive I'll define a value called mapping which is a Java Script object that maps values of the highlight prop to the corresponding background color. So if highlight is none then the background can be the empty string. If highlight is correct then we want the background color to be green and if highlight is wrong then we want the background color to be red and then finally to make that function work we return the correct value for the highlight. Now instead of hard coding the background color to white, we'll call our new function passing in the highlight value. With that done our turn component is now expecting an extra prop. Turn is used within the Author Quiz component. So here I have to provide a highlight prop and it'll add highlight as a prop on the Author Quiz component as well. So that's just passing straight through Author Quiz. Going up one more level where the Author Quiz component is used, we can add a highlight value to our application state and let's see what happens if I set the highlight value to wrong. I've opened the console and now I'll start the application. You can see that background of the application has changed to red. If I set highlight to correct, then the background of the application changes to green and if highlight is the empty string the background of the application resets to white. So the read side of highlight seems to be working fine. If we pass in a value to our Author Quiz component then the background color is set correctly. But what we haven't managed to do is respond to the users interaction with our game. And the way that the user interacts with our game is by clicking on one of the potential answers. We'll start by adding a onClick event handler to our book components. Each of these is a book component. We'll add another prop to book which we onClick, onClick of the function. When a user clicks the div containing the book title we will call the onClick prop and we will pass title of the book. Because we've added a onClick prop to book, we need to see where we're using the book component and make sure that we're supplying that prop. And this is where we'll take the opportunity to convert what is essentially a DOM event onClick into a component event. And you may recall that that is an event that's expressed in the language of our domain model. So instead of onClick, I'm going to call the function onAnswerSelected because the point of view of our

application domain, that what's happening. At the low level it may be a click on a div but conceptually the user is selecting an answer to the quiz. An answer selected will be a prop of the term components. Which means that it must be provided by the Author Quiz components. Once again, the same as we did for highlights we'll simply pass the onAnswerSelected option straight through the Author Quiz which means that it's added as a prop of the Author Quiz onAnswerSelected. Looking at where our Author Quiz is used, we can finally provide an implementation for onAnswerSelected. So now we need to stop and think about what we want to have happen when the user selects and answer to the quiz. Well the first thing is we need to work out if their answer is correct or incorrect. So let's create a value called isCorrect and to assess that I need to inspect the turn data books collection and see if I can find one book in that collection such that the title of the book is equal to the answer the user selected. So we're looking at all the books belonging to the author displayed and trying to see if one of the books is the book selected by the user. And if that's true then they've selected the correct answer or a correct answer. Based on that correctness value it sets the highlight. If their answer was correct, then we'll set it to correct. If their answer's incorrect then we'll set it to wrong. We've handled the DOM event, we've handled the component event, we've updated our application state. But none of that will have any effect on the UI because the application's not being updated with that new state. And ultimately what needs to happen is that we need to tell React that the application needs to be rendered again. The way that I'm going to do that is that I'll put the rendering into a function. The very imaginatively titled render. Need to make sure that render is called when the script is executed but also, we'll call the render function after we've updated the application state so that that state change will flow through our user interface. Let's try that now. Okay so I'll select an answer. I selected The Shining which was not written by the author shown. So my answer was incorrect and the application has it highlighted in red. Now if I select Heart of Darkness, Heart of Darkness was written by Joseph Conrad who is shown here. So the answer is correct and the application highlights in green. Going back to our turn component, its props object has become quite complicated. And the correct functioning of our application depends upon those props being supplied correctly. So this is a good opportunity to add some prop type validation. And the way we do that is by adding a prop types property to our turn function. Prop types is an object which has a key for each prop. First prop is author and author is an object. So I'll use the prop types shape validator and this lets me specify what that author object should look like. It should have a name. A name is a string and it's a required value. ImageURL is also a required string as is imageSource. The books property of the author object is an array so we use the array of validator and it's an array of strings and that array is required. Other props of our term components include books. Same again. We have the onAnswer selected function which we can validate with the

PropTypes. func validator. We also have the highlight prop which is a required string. Adding that prop type validation is giving me an error which says prop types is not find and that's because the prop types value has been moved out of the core react. So if we want to use it we have to import it separately. But often in my react import I'll add a second import for PropTypes and they come from the PropTypes module. Now the application is running again and working.

Testing the Author Quiz

When we last tried to test our Author Quiz application, we weren't able to get very far because the application didn't do very much. However we've since added more complicated rendering and some interactivity in terms of being able to handle the users input when they click on a book and make the user interface update accordingly. So now's a good time to see if we can uncover some of that functionality with our test. To be able to test components easily we will once again restore enzyme. Now we can add some imports for enzyme, we'll import the adapter for React 16 and configure the adapter. So that is essentially connecting enzyme to the enzyme React adapter. With that in place we can start our testing and our existing test is actually broken. The AuthorQuiz. test. js. Suite. Click to failure. We can see that the failure is in the prop types. The turn component expects a books prop because it's marked as required but it's not being provided. And this shows some of the benefits of adding the prop type validation. It's still a run time error but now we're get a much more helpful error message. The Author Quiz component expects our application state to be provided which we're not currently doing. So, I'll add a value called state that has some dummy application state data in it. We've got a turn data, we've got the highlight value. So these are all the things that are required for the Author Quiz components and then we just need to provide that to all of the quiz elements. And Author Quiz also expects the onAnswered selected prop. And for the moment I'll just provide an empty function. With that done we're back to a successful test suite and well placed to start adding some more advanced tests. So within the Author Quiz test, I'll describe a more refined state which is when no answer has been selected. To set up the application to test this state, I use enzymes mount function to render an Author Quiz. Our scenario is that no answer has been selected so to go to this state check highlight is set to done. So we're already in the correct state and now we can add assertions. I would like to say that when no answer has been selected, it should have no background color. Using the wrapper that we rendered the component into, we can use its find method to find the turn div via ACSS selector. Grab its props, inspect the style prop and the background color property on the style and we expect that to be the empty string. A number of tests has gone up by one and all test are successful. If we wanted to be certain that that's working

properly we can change our expectation to be something that we know to be incorrect. Now the test fails and the test output is quite good within the Author Quiz suites and the states where no answer has been selected and the assertion should have no background color, we expected the value to be fuchsia but it was actually the empty string. And that's only going to do occasionally just to prove to myself that the test suite is working correctly and actually verifying something useful. We can go on and add other states now such as when the wrong answer has been selected. So this time because we want to test the state where the wrong answer has been selected, when rendering the Author Quiz elements you override the highlight value of the state to be wrong. And in that scenario our expectation is that the turn component would be rendered with a red background. So we add an expectation. It should have a red background color. The implementation is the same as before. We selected the turn div, grab out the background color from the props. Now this time we're asserting that it should be read and now we have nine passing tests. So that's working properly. For completeness then, I can copy and paste a whole group of tests and change the scenario to being when the correct answer has been selected the highlight value in that scenario is correct and the background color we should expect to be green. Now we have 10 positive tests. But what if we wanted to test an actual user interaction right down to the DOM click event level? Let's try writing a test for when the user selects their first answer. Once again we'll mount an Author Quiz component but this time I want to provide an implementation of the onAnswerSelected call back. So I'll create a value called handleAnswerSelected and SyntheticEvents the jest. fn method to create that function. Jest. fn creates a mock function. Now having mounted the Author Quiz component I can use the wrapper to find the answers which is the Dom elements with the DOM answer class. Select the first one and then simulate a click event on that helmet. Now we can add an assertion and I would like to assert that the onAnswerSelected component event is triggered. Another way to say that the onAnsweredSelected function should be called because handle answer selected is a special jest mock function we can use the expectation to have been caught. This means that the test will fail if our call back function has not been called. That test passes successfully. But, we can do even better than that. Because we don't just care that the function has been called, we also care about the value that's passed to that function. So we'll add another assertion. We choose that when the first answer is selected, the selected answer should be The Shining and that's because The Shining is the first book specified in our test data. This is another expectation against the handleAnswerSelected mock function and being a jest mock it allows us to use the two have been called with assertion to check that the argument passed to the function is what we expect it to be. The tests have run again and now we have 12 passed so that's all working correctly.

Summary

Events make applications fun. You could have an application without any events but it would be very, very boring. React provides two major features. A way of generating a user interface from a model and a system for processing events. So events are very important. The React way is simple and powerful.

Forms

Introduction

Most web applications end up requiring some support for forms. It's easy for service side web applications because HTML has a form element, which supports serializing and posting form values. Client-side web applications are left to implement form support for themselves. React has some very basic form support built in. This includes binding values to the form, binding values back out of the form and accessing individual form elements. Notably absent is any built-in support for form validation. React supports the usual selection of form elements. They function much the same way as their HTML equivalents but with some subtle and important differences. The way that React handles UI rendering has an important incompatibility with user input. There are good and bad ways of working around it. If you don't mind giving up a little bit of fine-grained control, you can gain a lot of productivity and simplicity by using a form library to automate common form handling requirements. Where there are forms, there is validation. React doesn't provide any assistance with form validation. But it's not difficult to implement. Finally should all else fail, it's possible to bypass React's form element obstruction, and access the underlying DOM elements directly. Use with caution.

Form Elements

React allows the programmer to work with form elements such as the various inputs, text areas, and select lists using a syntax similar to html. When React renders a form input, it must preserve React's rendering semantics, which guarantees that the rendered UI is a direct translation of the user interface model. This has the effect of preventing user input. Obviously a bit of a problem for a form. We will see how React solves this problem later. But first let's have a look at some common form controls. The text input matches its html equivalent. The content of the input control is set using the value prop. The text area element is noteworthy, because it differs slightly

from html. A HTML text area provides its content as text between the opening and closing tags. A React textarea uses a value prop just like the text input. Select is another element that is nicely matched to its HTML equivalent. A HTML select list is a select element containing option children. The selected option is set by providing a selected attribute to an option in the list. The React select element is similar except that the selected option is set by providing a value prop to the select element. The option with the value matching the value set on the select element is selected. This React component demonstrates the form inputs we've looked at so far. Firstly, we have a text input with the value set to react, which renders as we would expect. Similarly a text area with the value set to react and finally a select list. The select list has two options, Saturday and Sunday and the value is set on the select element to Sunday. And you can see that that is the option selected in the list.

Allowing User Input

Form elements preserve React's rendering semantics, which binds the state of the user interface to the state of the model. Allowing the user to input values into a form element would break that model and abandon the simplicity that is React's great strength. Therefore form elements by themselves are read only. A simple way to allow user input without violating React's rendering semantics is to use React's component state. A form by itself as shown here is read only. The user cannot change the content of the text inputs. This is because the value props for the text inputs are defined to be the empty strings for this component. They cannot change. This is the basic React form component. I've called this component identity, it renders nothing but a form containing two text inputs. Each text input is bound to the empty string. And that form is rendered, if I try to type into those text inputs, nothing happens. I'm unable to enter a value into the text input because doing so would violate the definition of the component, which says that the value of the text inputs is the empty string. If it starts as empty string, it must always be the empty string. To allow a user to provide input into a form, first we add state to the component containing the form. Then bind the value of the inputs to the component's state. This is very important because the values of the inputs are bound to the state. They can change if the state changes. What made our read only form read only, was that the values were bound to empty strings and could therefore never be anything else. To allow the input values to change, we must bind them to something that can change. The final step is to add onChange event handlers to catch the user input and update the component's state. Because the content of the form is bound to the component state, when we change the component state, we also change the content of the form. And therefore users can enter values into the form elements. This is a form similar to the

previous example but updated to allow user input. It's still a component called identity. It's still declared using the JavaScript class syntax and I've done that so that I can use components state. In the constructor of the component, the state is initialized to an object with two properties, first name and last name that are both set to empty strings. The random method for the component returns just a form with two text inputs. The differences are that this time the value is bound to the properties of the component's state. And we do that so that we have a mechanism to allow those values to change. We also have a handler defined for the onChange event called onFieldChange. onFieldChange uses the name of the text input to update the state of the component using the set state method. And the state for the name of the input is set to the value of the input. The other thing we had to do in the constructor was to bind the onFieldChange method to the class. And that is necessary so that within onFieldChange we can access this and have it referred to the component. Now in effect what happens, the component is rendered. Then a user can type a character into one of these inputs, which triggers the onChange event for that input, which updates the state for the component, which causes the component to be re-rendered and the character they typed is the value of the input. And that is how we allow user input in a React form.

Form Libraries

Entering data into forms is a common and well understood user interaction. Creating forms, making them accessible and implementing good validation can require a lot of effort and maintenance. There are many form libraries available for React. Typically they provide a more efficient declarative way of specifying forms and help the programmer connect JavaScript data to the user interface form. Using a form library is a compromise. You give up a degree of control in exchange for increased productivity and reliability. For demonstration, we will look at a library called React JSON schema form, which is available as an NPM module. React JSON schema form is a library that uses JSON schema to define the data model of the form. JSON schema validation is a draft standard providing a way of defining the allowed values of a JSON document or JavaScript object. Because it includes a lot of information about the data structure and values it may contain, JSON schema provides a good starting point for both form rendering and form validation. It's also nice to use a standard so that the validation rules once defined can be used in other places to validate the form data. This is the demo site for React JSON schema form. On the left, we have the three inputs required to create a form. At the top the JSON Schema. So this is the schema that defines the format of the data that is bound to the form. So you can see that it specifies which elements are required and it also specifies all of the different properties of the

object, what the data type is. And any other validation rules. For example the password that's specified to be a string with a minimum length of three. Further down we have the UI Schema. This is a way of overriding the default rendering for a property. For example, it says that when we come to render the bio property, the UI widget to use is the textarea and you can see the Bio property is indeed a textarea. Finally the third required input is form data. So this is the initial JavaScript object that is bound to the form. So first name Chuck and the first name input has a value Chuck. Let's try a demo of our own. I have imported the form object from JSON Schema form. And next I've defined a JSON Schema for my form. My form will have three properties. First name, last name, and age. First name and last name are strings. Age is a number and for the first name field, I've specified that the first name must be between one and six characters in length. First name and last name are required meaning that age is an optional field. Having my schema, I can now use the React JSON Schema form form component to create my own form. We have to provide the schema. I've disabled HTML5 validation and I'm handling the onSubmit event. And just sending whatever the result is to the console. And here's the form that it's generated. It has three text areas for my three properties. Given that age is a number, if I wanted to I could have provided a UI schema to override the control that's used for that property and I could've used an up down control or a select list or some other control, if I wanted to. Now if I submit the form, I will get two fields highlighted as being an error. First name and last name, because they're both required properties. Put a value in for last name and put a value in for first name. Now I want to submit the form again. Just first name is an error and that is because first name was defined as between one and six characters and I have provided more than six characters, fix that up. If I put a non-numeric age in, that also is validated because our JSON Schema specifies that age must be a number. Okay so I bring up the console, submit the form, validation errors have gone away. And the object passed to the onSubmit handler, includes the form data with the age and the first name and the last name. So this is my JavaScript model produced from the form data that has been validated with JSON Schema. If I submitted the form with an error, then the object passed to onSubmit would look quite different and it would include details of the validation error that occurred.

Form Validation

The easy way to add validation to a form is to use one of the form libraries that has validation included. If you require complete control of your forms, and building them by hand then you will need to implement validation too. Users need to see what they type. So you'll still need to synchronize the form user interface with the model object is generated from. You then must

decide when to trigger validation. You can validate on each change to the form. You can validate when the user moves between form fields or you can validate when the form is submitted. Once validation is complete, you must decide how to feed the results back to the user so they can correct any validation errors. It's common to show errors in line next to the field having the error or to show errors altogether for the form or some combination of both. What you choose is mostly about what will work for your user experience. And how much effort you want to put in.

Client-side Routing with HTML5 pushState

HTML5 includes an API called pushState or history that allows JavaScript to update the browser URL without triggering a request to the server. An application running in the browser can cause the current URL to change, then detect that change and handle it. If the user manually modifies the URL that too can be detected and handled in the same way. The great thing about using pushState for routing is that the URLs are indistinguishable from normal URLs used with server side applications. The effect from the user's perspective is that the application behaves exactly the same way that the server-side application would. Preserving the URL based nature of the web. Implemented properly, this enables proper refresh behavior bookmarking and the browser back button. Because the URLs are proper URLs, the server needs to be able to respond to the request. One technique is to have the server return the same page for all requests and then let the routing happen client-side or if you want to allow your application to be crawled by search engines, you can have the server return rendered pages. React Router is the most popular client-side router for React. It provides conditional rendering based on routes. Meaning that when a route is requested React Router can take care of rendering the correct component or components. It also supports route parameters, so components can receive data based on the route. Finally React Router provides a convenient way to update the current URL path in turn triggering the router to update the user interface to match. The React Router route component is the primary tool React Router provides for routing. You place a route element somewhere in your application with a path prop and a component prop. If the current URL matches the path specified then the component specified is rendered at that location. The example shown here understands two URL paths. Forward slash and forward slash about. If the path is forward slash, then the welcome component will be rendered in this location. If the path is forward slash about, then the about component will be rendered. No other possible path will be matched. The exact prop changes the way that the path is matched. If exact is specified, then the path is only matched if the current location is an exact match for the path prop. If exact is not specified, then the match is more of a begins with. For example, because this route requires an exact match on

the forward slash about's path, it will only be matched if the URL path is exactly forward slash about. If the exact prop was removed, then this route would match any path beginning with forward slash about, such as forward slash about forward slash get. The route component also supports path parameters. By prefixing a path segment with a colon, we specify that the segment is a variable. The value supplied in the colon id position, will be assigned to the id parameter and made available to the single component. The single component receives a match prop, when it's rendered due to a React Router route element. The match prop has a params object, which contains any path parameters. In this example, id. Here is a more complete example. We can have as many route elements as we like. Each of them attempts to match their path prop to the current route path. If they match, then they render their component. The first route element will match any path. So it's redundant. I could replace it with just a menu element. I used a route element just to demonstrate that you can have route elements anyway you like. Within the detail section, if the route path is exactly forward slash, then the welcome component will be rendered. If the route path starts with forward slash list, then the list component will be rendered. If the route path starts with forward slash single forward slash something. Then the single component will be rendered and the id parameter will be supplied. Where route elements occur, there must be a browser router element somewhere higher in the component tree. This is a consequence of how React Router is implemented. Not a logical requirement. Here we have a simple application. The UI consists of a long list of links and a grid of four squares. Each URL indicates a grid position in the first example top left, it indicates a character and it indicates a color. For example if I choose bottom left lady-glasses/Green, I navigate to that URL and my application is updated with a lady wearing glasses shown at the bottom left in green. Or I could've chosen top right tie-guy/Orange. In addition, there's a little bit of text at the bottom that says something at the top if the character is rendered in one of the top cells and says something at the bottom if the character is rendered in one of the cells at the bottom. The routing in this application is done with React Router and to do that I import three identifiers from the React Router DOM module. We have the browser router, the route and the link. Browser router and route we've already seen. Browser router is the wrapper around any elements that want to use routes. Route is the component that we use to conditionally render other components based on the route path. And link is a component that we can use to generate a link, that will update the path for us. Next up we have a little bit of data definition. So this is where we define the possible characters that can be rendered. There's two at the moment. There's lady-glasses and tie-guy. And for each of those there is a URL for an image of that character. We have a React component called character and this is the one that takes care of rendering the image for a particular character. It expects a match prop having params for the character and the color. And it generates an image URL from that and renders it inside a div.

Dashboard is a React component with the four cells and each cell includes a React Router route component. Routes beginning with top left will match the route specified in the first cell. Routes beginning with top right match the routes specified in the second cell and the same for bottom left and bottom right. In each case, it's the character component that is rendered which is matching different paths in different locations in the DOM. Finally where we render our application using React DOM to render. The whole thing is wrapped in a browser router because we have to do that to get the route elements to work. On the left hand side of the page, we take the characters specified earlier in the code. And we map them. For each character, we take an array of cell positions and we map those as well and then for each combination of character and position, we map the four possible character colors, gray, green, orange, and purple. Having done all that, we generate a link. The link begins with a cell position followed by the name of a character, followed by a color. We might have top left tie guy purple for example. When a user clicks on that link it will coerce the URL to be updated with the path that we've generated. On the right hand side of the page, we have the main section, which includes three routes. Firstly, we have a route matching the path forward slash. This is not an exact match meaning that this component will always be rendered. So that's how the dashboard component gets rendered. Next up we have a route with the path forward slash top. And this renders the text something at the top and I added this one to demonstrate how partial route matches work. So any route beginning with forward slash top will match this route. So that could be top left, it could be top right. It doesn't matter, if it starts with forward slash top then we will get the output something at the top. And similarly for forward slash bottom. Any path starting with forward slash bottom will include the text something at the bottom in this location. And that is how the application is put together.

Adding Routes to the Author Quiz

I'd like to add a form to the Author Quiz but to get to a form I need some navigation and that means URLs and routing. But this is a client-side app. So how do we do routing? For the Author Quiz application, I will use the React router library to add some simple routing. We need a home route and I'd like to add a new feature to the application. The ability to add a new author. We will need a new route that will render a form allowing us to enter the data for a new author, which includes their name, the URL of an author image and a list of their books. I have opened the Author Quiz application in Visual Studio code. The first thing that I need to do to add routing to this application is to install React Router. And the way I do that with the terminal is npm install and the package is called react-router-dom. Everything that we need is in that package. While I'm

waiting for that, I'll open index. js and this is sort of the route of our application. At the top of the file, we need to import the React Router library and we'll import two values, browser router and route. In the function that renders the application, we currently render an AuthorQuiz element. The first thing I'll do is create a new React component to render that element for us. And I'll just call that App because it's the top level of our application. It simply wraps the Author Quiz element. Now I can change this to app. I've made some significant changes so I'd like to just check that the application is still working and that looks fine. Now let's add some routing. So the first thing to do is wrap our components in browser router and that gives us the ability to introduce route components. So the first thing I'll do is add a route for the route path. I'll make this an exact route matching the route path and it should render the component app. If I've done that right, I should have preserved exactly the behavior that we already had because what I'm saying is that the router should render the app component in this position if the route path matches the route path exactly. And my application still works. The purpose of adding a router was to be able to add a second route. That second route will be a place where we can add a form that the user will use to add new authors to the application. So I'll create the second route. This time we'll match the path forward slash add and when that route path is matched, render a component called AddAuthorForm. That has broken the application. It no longer compiles because AddAuthorForm is not defined. It's correct because that is a component that we haven't created yet. So I'll add a new component AddAuthorForm. We have a header, Add Author, and for the content of this component, later we will add a form obviously for adding new authors. But for the time being we're just working on routing. So for the content I will just dump out the prop data that we get from the router and the router supplies a prop called match. We have another error. It says that the router may have only one child element. That's absolutely correct. So it's complaining that I've defined two routes as direct children of the browser router. So we need to wrap them together into a single parent. And this is where React fragments come in very handy. So it lets us solve that problem of grouping React elements under a single parent but using a component that has no DOM representation. So by the time the application is all rendered out, this React fragment component doesn't add anything to the DOM. If I'd used a Div or something of that nature, then obviously I would be adding another element to the DOM for no purpose other than to satisfy one of React's requirements. So the application looks like it's working. Let's try our new add URL and that's the routing working for us. So that's taken us to our Add Author form component. It's our Add Author header and then the content of the page simply being the route data that we get from React Router. It's telling us that the path is currently forward slash add and it's an exact match with no parameters. If I use the back button, it will go back to the main interface. I'd like the Add Author form to be discoverable. So let's add a link to this page. We

need to go to the AuthorQuiz component and between the Continue button and the Footer I'll add a paragraph containing a link. So this is our first time using the React Router link component. So before that will work, we need to import that. So we import link from react-router-dom. The link component expects a to prop and this is where we specify the path. We provide the link text as a child. We'll say Add an author and then close the element. So now our AuthorQuiz application has this link add an author. When I click it, it navigates to the add URL. And that's all that we have to do to add routing to the AuthorQuiz using React Router.

Refs

Ref's are a way of accessing the underlying DOM elements that are wrapped by React components. So that you can imperatively modify them. They break React's declarative rendering power dom, which is why they're useful. It's also why they're a tool of last resort. To date there have been four different APIs for this purpose. They are find DOM node, get DOM node, call back refs and the current createRef function. The React's documentation recommends the use of createRef. It's common to create refs in the constructor of class components, and design them to the element. Refs are created using the React. createRef function. Then in the render method, use the ref prop to associate a DOM element. In this case a Div with the ref we created. Following the render the current property of the ref points to the DOM element. So the DOM element API is available such as the inner HTML property. Here I'm using the componentDidMount lifecycle method which is a common place to take advantage of refs. Because it is called after the component has been rendered, and the DOM elements have been created. This example demonstrates the usage of createRef. As per the slides, I have a React component created using the JavaScript class syntax. In the constructor, I create a ref using React. createRef and assign it to the myDiv property. Then in the render method, I use the ref prop of a Div to associate the Div with the myDivref. Having done that, I can access the current prop of myDiv in componentDidMount to get a reference to the DOM element that is rendered for my ReactDivelement. And having access to the underlying DOM node, I can access its inner HTML prop and here I am appending a string of HTML. Because I'm appending a string of HTML to a DOM node, this content is not escaped, and will be interpreted by the browser as HTML. In the render function, I included a very similar message that also is a string of HTML. Because this string of HTML is provided as a child to a React div element. React will take care of escaping it for us. This string won't be interpreted by the browser as HTML. It will be encoded so that it appears in the UI as HTML. And you can see what I mean here. So when that string is rendered, it appears as HTML. Conversely, the output generated by appending to in our HTML, is interpreted by the

browser as HTML. So this is a dangerous way of inserting content into the DOM because if this string of text came from a user, they could potentially insert some malicious code in there that would be executed by the browser. However, it does demonstrate the usage of ref and createRef to get access to underlying DOM nodes and manipulate them with the browser API.

Adding a Form to the Author Quiz

It's time to return to the AuthorQuiz. Previously we added routing, and created a place for a form to let the user add an author. Now it's time to implement the form and update the application to use the extra data. I've opened index. js, which is the module containing the root of our application. Here's the routes we added last time and if the path matches forward slash add, then we render the AddAuthorForm which is a component defined just above. To make this easier to work with, the first thing I'd like to do is extract AddAuthorForm out into its own module. So I'll create a new file called AddAuthorForm. To be able to work with JSX we need to import React. And now I can move the AddAuthorForm component into the new module, and I'll make the component the default export from the module. Okay now in index. js, we need to import the module we just created. And hopefully everything will still work. Looks good. Switching to the AddAuthorForm module, we can get rid of the route data that we were rendering and replace it with a form. Inside the form or add a label for an input called name and note that the label has a prop htmlFor. This is because JavaScript defines for as a reserved word. So in JSX they have to use something else. Next we'll add an input. And that should give us a form, and so it does. There's our label and there's our text input. The form doesn't look the way I'd like. So I'll add some styling. I'll add a class on the addtoDiv and on the div wrapping the element. Next I'll add another file and this one will be style shaped for the AddAuthorForm component. And it's called AddAuthorForm. css. I'll add some styling in there just to make the form look the way that I want. And then back in AddAuthorForm, I need to import that style sheet. And the web pack setup in createReactApp will pick up that CSS import, do some magic, and make it all work. So you see that the form looks a little bit different and hopefully a little bit better. The purpose of this form is to add an author. So I need a few more inputs, so the next one I'll create is for the imageUrl. Now I would like to bind these inputs to some component state, but because my component is a function, I don't have access to React component state. The way that I'll solve that is to extract the form part of the component into a separate component. So I'll create a new React component called AuthorForm, give it a render method and move the form markup up to my new component. And that still works, excellent. So the point of doing that was to bind the inputs to a component state. So I'll create a constructor. The constructor takes some props and passes them

to super and then we can create our default state for the component. We need a name, property. We need an image URL property. And now I can bind my inputs. So I specify that the value for the name input should be this. state. name. And for the image URL, this. state. imageUrl. I'm now unable to type into my form fields, because I have declaratively specified that the value is the name property of the state object, which is declared in the constructor and never changed. So I need to add a way of changing the component state. Earlier, we saw that we can do that using this method onFieldChange. In the input control, we add an event handler to onChange and set that to onFieldChange. Because onFieldChange uses this, we have to make sure that in the constructor, we bind this. What this does is it guarantees that no matter how onFieldChange is called, the value of this within the method will be the same as the value of this within the constructor. And I'll do the same thing for the image URL. Now we can type into the text inputs. I suppose the next thing is to be able to submit the form. To do that, we use the onSubmit event handler. And I'll bind that to a handleSubmit method. The first thing to do in handleSubmit is to call the preventDefault method on the event. And that's to stop the form from being submitted. Once we've done that, we can call a method on the component props, which I'll call onAddAuthor and I'll just pass out the current value of the state object. onAddAuthor doesn't exist, so we'll go to the parent of our author form, which is AddAuthorForm. I'll add it as a prop here. And pass it through to the AuthorForm. Then in the parent of AddAuthorForm in index. js, we can supply an onAddAuthor handler. To give me a place to specify a prop for my AddAuthorForm, I'll create a simple wrapper function called AuthorWrapper. That simply wraps the AddAuthorForm element, and gives me a place to specify the onAddAuthor prop. For the moment I'll set that to console. log, just so that I can see that everything is working correctly. And note that this is the same trick that I did with app. So I simply created a little intermediary component to give me a place to specify props. And now we change the component used in the route. Let's try entering an author's name and image URL. Oops, I need a submit button to be able to submit the form. We go back to the AddAuthorForm module and the AuthorForm component. At the end of the form, we'll add another input. This one will be a submit button with a label add. When I submit the form I get an error. In the handle submit method, it's saying that this is undefined. Hopefully, you guessed that that is because I forgot to bind this in the component's constructor. So I need to add a binding, like so. We'll try that again. Okay and now in the console, when the form is submitted, I receive an object with a name and image URL matching the values I specified in the form. There's one more thing that we need to create an author that we can add to our application and that is a collection of books. In the state for the AuthorForm, I'll add the required books property and initialize it to an empty array. In the render method, I'll add another input section for the books and within that section, the first thing that I want to do is render the current contents

of the books array. So each book will be mapped to a paragraph containing the name of the book. It doesn't look like much at the moment. Just for a test, if I go to the initial state of the component, and add a couple of book names, then I can see that those books are being rendered into paragraphs. Also note that I get a warning saying that each child in an array should have a unique key property. That's easily fixed. That's just saying that these paragraphs need to have a key prop set to something unique. So I'll use the name of the book. Now the error has gone away. Next I need a way of adding new items to the book collection. I'll create another label input pair this time for a value called bookTemp. There's the label and there's the input. The input is bound to a bookTemp property on the state. So I have to add that. Initialize to an empty string of course. Now I have my bookTemp field that I can type in. I'll move the label above the list of books and then add a button and this is the button that the user will use to add the book they've typed into the input to the collection of books. When the button has clicked, we'll handle that event. So we need a handler, which I'll call handleAddBook. handleAddBook once again needs to be bound and then within that handler, I can call setState to update the component state. We already have a books array, but I want to update it to its current value joined together with the current value of bookTemp. At the same time, I want to clear the bookTemp value. So whatever is in the bookTemp input, gets moved into the books array and then bookTemp gets cleared. I'll get rid of that test data in the state. Okay let's try this. I've typed a book name into the bookTemp field and I'll click the add button and that book is moved into the books array. And I can continue doing that with as many books as I like. Now if I submit the form, the object that's passed out of my component contains the name, the image URL and the books array, which is everything that we need for an author to add to our application. Switching back to index. js, and our own AddAuthor event handler. I no longer want to send that data to the console. Instead I want to do something with our newly defined author. Our existing set of authors is stored in an array called authors. So the new author can be pushed on to the array, but now the problem is that the user has filled out the form. They've clicked the submit button. The new author has been added to the application but the user is still sitting there looking at their completed author form. What we want is to then redirect them back to the main application. Since this is a client-side application, we can't actually trigger the browser to change the URL. Instead what we want to do is navigate with our client-side router and the way that we do that is by going back to our imports and importing another value from ReactRouterDom called withRouter. withRouter is a function that allows us to give components access to a number of useful values but the one that I'm interested in is one called history. I'll convert author wrapper to constant and set it equal to the result of the withRouter function and withRouter is a function that expects one argument which is a React component. And that component has a prop called history. Now that I have access to history, after I've added

the author to the author's collection, I can use history to push a new path to be root of the application. Okay let's try adding our new author. Provide a name, an author image, and a collection of books. And then finally submit the form. Now we are back to our main application. When I play the AuthorQuiz, if I get the answer right, we'd like a continue button to appear that when clicked will load the next round of the game. The code has a placeholder for the continue component but it's not currently implemented. The continue component needs two props. A Boolean value show, which determines if the button should be visible or not. And a function onContinue, and this is the event that gets called when somebody clicks the button. Then in AuthorQuiz where we use the continue components, we provide those two props. The continue button should be shown if highlight is correct and the onContinue prop should be another event handler passed straight through. So I have to add it to the AuthorQuiz props as well. Then go back to index. js and we need to provide an implementation here. When the continue button is clicked, then we should reset the application state and rerender the application. Reset state doesn't exist. So we need to create that. Move our default state definition into the resetState function. Now I get an error saying that state is ready only. So we cannot assign it to something else. So I need to change that const to a let. Okay so now when I select the correct answer, the continue button appears. Clicking continue loads new game. The only thing I don't like is that we just need a little bit of padding around that button, which is easily fixed with a bit of CSS. It's better. Okay let's try out our Add Author form. I supply a name and image URL and a collection of book titles. Now submit the Add Author form. Hopefully that has added our new author to the author's collection. And because the turns are generated randomly, I may have to go through a few to see the new author. There we go, I got it on the second attempt. This is the new author that I've added. That was the author image I specified. And Clear and Present Danger and The Hunt for Red October are books by that author. So Hamlet and Romeo and Juliet are wrong. Clear and Present Danger and The Hunt for Red October are correct answers. Now our AuthorQuiz application uses routing to route us to a different view in the application. And we've used a form built with React to add a new author to our application.

Summary

Because React is most useful for interactive web applications, they often involve forms. React itself includes well designed primitives for working with forms. But doesn't make any attempt to provide high level abstractions or productivity. If you need to do a lot of work with forms, it's worthwhile exploring the available React form libraries. You should be able to find increased productivity and consistency.

State

Introduction

User interfaces are notoriously complicated to implement. To provide the experience that users want and expect, they must deal with many orthogonal concerns, as well as asynchronous and event-based processing. The permutations of possible pathways are effectively infinite. It's easy for programmers to dig themselves into terrible holes when creating user interfaces. The implementation of user interfaces are often difficult to understand and nearly impossible to maintain. In the React world, a number of ideas and patterns have emerged that help to simplify the implementation of user interfaces, so that it is possible to implement a user interface that is well-structured, obvious, and understandable. The first of these ideas is that a user interface should derive entirely from an application state. The second is that the application state should be externalized in a single source of truth and modified via carefully controlled mutations. Managing application state is a simple idea, but the implementation can easily become unnecessarily complex. The layers of optimizations that are added onto mature applications obscure the true simplicity of the ideas. For this reason, this module will introduce state management for React application by building up from first principles. I aim to show that the important ideas are straightforward, and the more advanced techniques may optionally be added if and when required. Before we can talk about how to manage state, we must understand the architecture that state containers exist to facilitate. Simplicity is hard. And to achieve it, we must have a clear vision of the architectural ideal we are striving for. We will learn about the model-view-intent architecture and implement a simple example using plain JavaScript. A state container is a tool for centralizing application state. For React, this means user interface state and controlling how changes are made to that state. Once we cover what a state container is, we will build a simple one from scratch. Just because you can build your own state container doesn't mean you should. The React eco system has a very good one, called Redux, that exist and is popular. Moderately complicated React applications will often benefit from something like Redux. We will build on the previous example, replacing our custom state container with Redux. While Redux is primarily made for and used with React, it's actually a standalone state container that can be used without React. If you are using React and Redux together, then there is a separate package called React-redux that makes it easier to work with React and Redux in the same application. Once again, we will update our example, this time adding React-redux.

Model-view-intent Architecture

You have probably heard of model-view-controller, a common pattern for server side web applications. When web development first begin to move to the browser, many different libraries attempted to port the model-view-controller ideas into client-side frameworks. Angular 1 is an example of this. Over time, developers learned that model-view-controller didn't work well on the client side, and new ideas gained popularity. One successful architecture for client side web applications is called model-view-intent. It's the architecture of well-designed React applications and also appears with other libraries, such as Elm and Cycle. js. An application built with the model-view-intent architecture is described by three components. The model, the view, and intents. The model is a single object that completely describes the state of the user interface. The view is a function that transforms the model into the user interface. That is the model is the input to the view function and the user interface is the output of the view function. At any moment, the user interface can be generated based on nothing but the model. When the model changes, the view function can generate the corresponding changed user interface, which leads us to the question, "How is the model changed? " the user interface generated by the view function can produce intents. Intents are things the user wants to do. In our Author Quiz application, the user selecting an answer is an intent. When an intent is produced it is applied to the model creating an updated model. The updated model is then passed through the view function to produce the updated user interface. This process forms a neat predictable cycle that is easy to reason about for a few key reasons. One, the model is the total source of truth. The entire user interface is described by the model. Two, the view produces the user interface based on nothing but the model. And three, the model can only be changed by processing intents on the current model. Model-view-intent can be thought of as a finite state machine in which the model is the set of possible states, and the intents are the possible transitions. Intents transition the model from one state to the next. The view is a function from the model to a user interface. If we were to build up a model-view-intent architecture from scratch, we might start with the following pseudo code. The model is a plain JavaScript object. It will contain properties according to the complexity of the view. The more the view can change, the more those possibilities must be represented in the model. The view is then a function that takes the model we have made and produces a user interface. In React, this user interface can be represented with JSX. The third component, update, is the function that controls how intents form the user interface are applied to the model to produce an updated model. To demonstrate the MVI architecture, we will build a stopwatch application. It will display how long the timer has been running in minutes and seconds. It will have a button to start the timer and a button to stop the timer. Just like in these slides, our MVI

example begins with a model object that is intended to encapsulate the entire state of our application. We're building a stopwatch. So to begin with our model has two properties. Running is a Boolean value that indicates if the stopwatch is running and time is the current value of the stopwatch. Next we have our view function, which as we've discussed, is a function that can take the model at any moment in time and convert it into a user interface. In this example, I'm using React and JSX to specify the UI. And at the moment, all it does is render a div containing the time value of the timer. And finally, we used ReactDOM. render to render the result of passing the model to the view function, because the time in our model object is zero, that's what we see in our user interface. The model view intent architecture uses user intents to update the model. And updating the model in turn updates the user interface. So, what are the possible user intents for a stopwatch application? I would define them here in an object. The user may choose to start the timer. The user may choose to stop the timer. The user may choose to reset the timer. There's a tick intent, and this one is not produced by the user, but it's rather produced by the environment. Because a stopwatch is measuring time, we use a timer to generate ticks once every second. This version has a more sophisticated view function. And now what it's trying to do is output the value of the timer in minutes and seconds. So the function firstly has to calculate how many minutes are represented by the time value, which is a value in seconds, and how many seconds remain once you've taken out the minute components, and then the output that's rendered is just updated to include the minutes and seconds values separated by a colon. Because the model now has a time value of 110, when the UI is rendered, it comes out as one minute and 50 seconds. Now I've made the view function just slightly fancier by adding a secondsFormatted options, and this is what's known as a left pad. The secondsFormatted value adds a leading zero to the seconds value if it's less than 10. So that the output is written as 8:01, not just 8:1. The other big change in this version is firstly that we now have a timer firing once every thousand milliseconds, once every second, and this is calling a function called update, passing in the current application state, as well as an intent. If we look at that update function here, the job of the update function is to imply the intent to the model and produce a new model, and how it does that is by having a map of a intents to functions that make some kind of an update to the current model. In this case, if the intent is a tick, then we want to take the current model and increment the time value by one, The final line of the function is simply selecting the correct update function based on the intent and then applying that function to the current model. Back in our timer, the new model is then reassigned to the model identifier and a render function is called, and a render function just wraps the standard ReactDOM. render where we render the output of the view function based on the current model. So now we have a complete working MVI architecture showing the completely life cycle. We start with a default model. We publish intent. An update function applies that intent to make some kind

of modification to the model, which then causes the application to be re-rendered through the view function based upon the updated model. Now we want to add some user interact to the example. The view function has changed to include a button, as well as the output of the timer value. It's a single button that's meant to be used as a start button if the timer is not running, and convert to a stop button if the timer is running when the application first starts the timer will read zero. And if a user clicks the start button, we want the timer to begin timing, and we want the start button to convert to a stop button. The way that we do that in React is by using the onClick handler of the button, which is bound to this handler function. The handler function calls the update function, passing in the current model and an intent. The intent that's passed to the update function depends on whether or not the timer is currently running. If the timer is running, then the intent is a stop intent. If the timer is not running, then the intent is a start intent. Now our update function needs to be able to handle those two extra intents. So the tick intent has changed slightly. Now instead of updating the model once every second regardless, we firstly check whether the timer is running. If the timer is running, then every second will add one to the time. If the timer is not running, then every second will add zero to the time. The start intent modifies the application state by setting the running property to true and the stop intent setts the running property to false. Everything else remains unchanged. So now if I click the start button, that publishes the start intent, the update function processes that start intent, changes the running value to true, and then on every tick, the timer is incremented. If I click the stop button, then the stop intent is published, the application is updated setting running to false, and the timer stops being incremented every second. And of course we can restart that timer or re-stop it as much as we like.

A State Container

Now that we are familiar with the model-view-intent architecture, we can look for ways to create abstractions that simplify the design. We already have React for the view component, converting the model into a user interface. A state container is a component that takes care of holding the model and controlling updates through the model. We will extend our stopwatch example to use a custom state container. The custom state container will implement three methods. GetState returns the current application state object held by the state container. Dispatch applies an intent to the application state, producing a new application state. Subscribe registers a call back to be called when the application state changes. That is when an intent is passed to the dispatch method. That's all we need for a simple state container. For this example, we will update our stopwatch to use a custom state container that we'll create in the process. The code I have here

so far is from the previous examples. So we have an update function, and it's responsible for being able to apply intents to the application state to produce new application states. The only way in which it's changed from before is that the initial value of the application state is now specified as a default for the model parameter. Similarly, we have the familiar view function. It calculates how to display the current timer value, and it has the timer value and a start stop button as the user interface that's generated. I've defined a value called container, which is currently an empty object, and this is where we will implement our state container. Other than that, things are basically the same. So, what do we need our state container to be able to do. Well, in the render function, the view needs to convert our application state to a user interface. So we need to be able to get access to our application state, and I'd like to do that by calling container. getState. So that should return the current application state when it's called. That's one thing we need to be able to do. If we look in our timer that's firing once ever second, we need to be able to dispatch intents to our container. So I'd like to be able to say container. dispatch, and the intent, this one is tick. That should work. And similarly, in our view, when someone clicks on the stop start button, that's another location where we need to be able to dispatch intents to our state container. If the timer is running, then we dispatch the stop intent. If the timer is not running, then we dispatch the start intent. That's a good start, but we would still have an application that doesn't actually do anything. We need to close the loop and make sure that the user interface is rendered when the model changes. And to do that, I would like to be able to subscribe a callback function to be called when the model changes. And in this case, that function is render. Every time my application state changes, I want to call the render function to re-render the UI from the current model and update the DOM. So far we've worked from the outside in and we've come up with the API we want to use to work with our state container. What we haven't done is actually implemented the container. It's just an empty object. We need a way of building our state container, so I'll call a function called createStore. This is a function that doesn't exist to create our application state container or store, if you prefer that terminology. To be able to create that container, because it needs to apply intents to our model, that create store function is going to need the update function. So now we can implement createStore. The argument passed to createStore is our update function, and we'll call that reducer for reasons that will become obvious a little bit later on. CreateStore needs to return our application state container, which is an object. And as previously described, it needs to have a dispatch function that expects an intent. It needs to have a subscribe function that expects a call back, and it needs to have a getState function that returns the current state. Now we can implement those functions. I'll start with dispatch. When dispatch is called and an intent is passed. We need to call our update function which we've called reducer. We need to pass in the current state and we need to pass in

the intent so we don't have that current state yet. So I'll call that internal state and define a variable with that identifier, internal state. Because the reducer returns the new state, then we need to assign that to internal state. So when somebody dispatches an intent, we set the internal state to be the new state produced by our update function. That also makes it trivial to implement our get state. That function can just return the internal state value. Subscribe is used to register callbacks that get invoked when the application state changes. So, when that function is called, we can simply push the handler onto an array of handlers. And once again, we need to define an identifier for that, and that can be initialized to the empty array. Now that we have that collection of handlers, when an intent is dispatched, after we've updated the internal state, we then need to invoke each of those handlers. So that's simply a matter of looping through the collection of handlers and invoking each one. That's all we need to do to implement our customs state container that has all of the features that we defined as being necessary. So, if we have a look at the output now, you can see that the application is being rendered, that's a good start. When I click the start button, the timer begins timing, and that is because the onClick event handler for the button is dispatching the start stop intents. Our custom state container processes those through our update function, the same thing is happening for the ticks that get dispatched once every second. The change in the model triggers the subscribe callback handler, which re-renders our application. And when we re-render the application, we pull the current application state from the container. So the functionality of the application hasn't changed, but now we're using a nice abstraction for the application store, which we've created ourselves from scratch.

Redux

As we have seen, it is not necessary to have a state container, and it is not difficult to build one. However, there are advantages to using a common state container library. It provides a degree of standardization. If many apps can use the same library, it is often less important what the standard is than that there is a standard. Using a popular library can also means that the implementation is likely to be more robust and have more features. In the world of React, Redux is a popular and quality state container. Most importantly, it provides a good basis for implementing the MVI architecture. Redux usage can be very simple, but it can also scale to sophisticated scenarios. Our custom state container used a function that converted the current state and an intent to a new updated state. In Redux, this function is called a reducer because it reduces the stream of intents to a single object, the application state at a moment in time. We have referred to the events that trigger state changes as intents, because that's what they're called in the context of the model-view-intent architecture. Redux refers to the same thing as

actions. An intent and an action are the same thing. From now on, I will tend to refer to them as actions, because that is the term used in the Redux documentation. The Redux API will be familiar. I used the same API for our custom state container. CreateStore is the function used to create a new store, which is the container for our application state. To create a store, the programmer supplies a reducer function and the initial store state. Get state returns the current application state from within the store. Dispatch sends an action to the store to be applied to the current state. The action is processed by the reducer function which builds a new application state. Subscribe registers a call back to be called when the application state held within the store changes. To get the benefits of standardization and depending on a robust and proven solution, let's convert our stopwatch to use Redux, instead of our custom state container, remember that the two solutions have a similar API, so we should not have to change much. This is the stopwatch application as we last left it. It has a view update function. It's using our custom createStore to create our custom state container, which is assigned to the identifier container. In this code pen environment, I have added a reference to the Redux script to make it available. And now what I need to do to convert the application to use Redux, instead of our custom store, is I'll delete the createStore function, because we won't be needing that anymore. And where it was called, let's change that to Redux's createStore. Now our application has disappeared, it's not being rendered anymore, and that tells me that there might be a problem. So look in the Console, and here I can see an error that says that actions must be plain objects. This is a Redux convention that says that actions should be objects where I'm currently using strings. Further part of that convention is that the objects should have a property called type. Now this is optional, but it is the way that Redux is typically used. So everywhere that I'm publishing an action, instead of using a string, I need to change that into an object with a type property. That takes care of the button click. We also publish or dispatch an action on a timer, and now my application is back. In the update function, I will rename intent to action because that's the terminology that Redux uses. And now instead of switching on the intent, I'll switch on the type property of the action. And that's all I have to do to convert the application to using Redux. Because our customs state container used the same API with dispatch, getState, and subscribe. The only changes that were required were to use Redux's createStore function and to convert our actions to be objects, instead of strings.

React-redux

React-redux is an extra module that helps with the integration of React and Redux. For sophisticated React and Redux users, it helps to make code neater and add some useful features. For beginners, it obscures the simplicity of React and Redux and make things more difficult to

learn. Make sure you understand React and Redux on their own before you look to use React-redux to integrate them. The main service provided by React-redux is to connect React components to the application state. React-redux can provide data from the Redux store to components when the component is rendered, and it can provide a way for components to publish actions that can then be used to modify the Redux store. Provider is a React component provided by React-redux. When it is included in a React application, it enables all React components below it in the component tree, that is its children or children's children, et cetera to connect to the Redux store. Connect is a function provided by React-redux that enhances React components connecting them to the Redux store in the ways specified. To specify what data from the Redux store should be provided to the React component as props, connect expects a parameter called mapStateToProps. mapStateToProps is a function from the Redux store to a set of props for the component. mapStateToProps took care of getting data from the store to the component. Another parameter of the connect function, mapDispatchToProps, takes care of specifying how the component can send actions to the Redux store. mapDispatchtoProps is a function from Redux's dispatch function to a set of props for the component. In practice, this provides a place to map component events to Redux store actions. In the last demo, we converted our stopwatch application from a custom state container to a Redux state container. In this demo, we will further enhance the stopwatch application using React-redux to decouple our React components from our Redux store. This is the example as we previously left it. So it's using Redux. createStore to create the application store. In my code pen settings, I've added another external library so that we can use React-redux. The way that React-redux works is by wrapping a Redux component in a call to the connect function, and in that function call, specifying how the component should be connected to the Redux store. In our example, we have a view which is a function from a model to some JSX> by definition, view is therefore a React component. And since it's a React component, I can wrap it in the connect function from React-redux. And as described in the slides, this is my opportunity to provide two functions, mapStateToProps and MapDispatchtoProps. Note that connect doesn't actually take the React's component as an argument. The two arguments are mapStateToProps and mapDispatchtoProps, but connect returns another function, and that function takes the react component as an argument. Now because view is a React component and we're using connect to convert it into a new React component, I will use the convention of an uppercase name for the component and call it stopwatch. Now I need to define those two functions. We'll start with mapStateToProps. And as its name suggests, it's the function from state to a set of props. So that's a valid implementation. And for the other one, mapDispatchtoProps, one again, the name tells you what it does. We have to go from dispatch to an object containing some props for our component, so again returning an

empty object as a valid implementation. Now we have to use our new stopwatch component, so let's go to where the application is rendered. Previously, we're calling the view function and passing in the application state. We'll get rid of that, and switch to our new stopwatch component. The way that React-redux connects components into the Redux store requires that those components be wrapped in a React-redux provider, so that the provider can make the Redux store available. So we'll do that, ReactRedux. provider, and we have to give a reference to the Redux store, which is called container. And now React-redux will take care of re-rendering our application when the state changes. So we can get rid of our render function. And also the subscription to the container, because React-redux is doing that for us. We just have to render the application the first time, and that's enough to get our application to render once again. When the stopwatch component renders, its data is supplied by the React-redux provider, mapping the store to the component via the mapStateToProps function. Currently, because we provide the empty object as the props, the stopwatch component isn't getting any props at all from which to render, and that's why the output is a bit odd. So here, instead of returning the empty object, we need to think about what transformation we need from the Redux store to props for our component. And in this particular application, the answer is no transformation at all. We simply want to pass the state straight through. More realistic examples will have more complicated application state. And only part of that state will be applicable to the stopwatch component. So we would use this function to extract out the pieces of state that we care about, but because this is a very simple example, the format of the application state is an exact match for the props that we need in the stopwatch component, so the function just returns the argument. Now when it comes to mapping dispatch to props, this is where we need to map our events that we want our component to be able to raise to dispatch and props. And there are essentially two events that come out of the stopwatch component that need to be able to modify the store, and those are starting and stopping which we will call onStart and onStop. So onStart becomes a prop that will be passed to the stopwatch component. We want it to be a function that dispatches an action. And onStart will of course dispatch the start action. onStop is very similar, but it will dispatch the stop action. So onStart and onStop then become props that are available inside of our stopwatch component. That means that in the click handler for the button, instead of dispatching actions onto the container directly, get rid of that, I'll also rename the argument to the component to be called props, just to make things a little clearer. And I can actually get rid of the handler function entirely, and just in the onClick binding for the button, switch on props. running. If the timer is running, then the event handler should be the onStop prop, otherwise the onStart prop. To get the application to work, there's one final small change we need to make, and that is in our update or reducer function. React-redux makes a

performance optimization where before re-rendering the application, it firstly checks if the application state has changed at all, and it does that by a reference equality check. Because this update function mutates the existing model object, React-redux doesn't see that as a change to the application state, because it's still the same object. So what we need to do is make sure that reducer function always returns a new object if the application state has changed, and I can do that just by providing an empty object as the first parameter to Object. assign, and that has our application working perfectly. So what have we achieved? Well, in the definitely of our stopwatch component, which is this part, you note that we've managed to remove any references to the container or Redux. The only dependency of this component is props. Its only dependencies are things that are passed in as parameters. The mapping of those props to the Redux state is done via the React-redux connect function and the mapStateToProps and mapDispatchtoProps functions.

Author Quiz State Management

When we last left the Author Quiz, the way that it handled application state was very simple. The application state data was defined outside of the React components and passed down the component tree by props. When an event occurred, it was passed back up the component tree, layer by layer, to be handled and applied to the application state outside of the React components. By introducing Redux and React-redux to the Author Quiz application, we can add a standard and robust way of dealing with state changes. And we can prevent the tedious requirement of having to pass data up and down the component layers. Note that this is a trade-off. Whilst it makes development much nicer not having to pass all data through all component layers, it also breaks the reusability and composability of our components. As the application currently stands, I can extract any component at any layer of the component tree and reuse it elsewhere without modification. Once we switch to React-redux, we can only move our components if we also move an equivalent React-redux setup. The simple presentational UI widgets, you are often better off using pure React component by themselves to get the maximum reuse and composability. For application-specific UI components that require complex bidirectional interaction with application state, you will often be better off using the React-redux connected component style. Now we will add Redux and React-redux to the Author Quiz application. The first step is to install the NPM packages. And when that's finished, we can add some imports for Redux and React-redux. Now that we have Redux installed, we can use it to create an application state container or what Redux calls a store. Creating a store requires specifying a reducer function, so let's start off with the simplest possible reducer. Now the

reducer is a function that takes the existing state and an action, and applies that action to the existing state to produce a new state. So the simplest possible implementation is simply to return the existing state. Now we have a Redux store, and we can use React-redux to connect to our React application. And the way that that is done is by wrapping any component that needs access to the store in a provider. And a provider has a reference to the store. The provider's job is to make that store available for all of its children and children's children. Just to try to keep the application working while we make the move to Redux, I'll leave the existing declaration of the state variable, and then we can slowly migrate to the new strategy. It's best always to keep your application working when you're doing a refactoring and working small steps and make sure that as you go, nothing is broken. So now we have a Redux store. It has no state in it and no way of mutating that state, and it's not used at all,, but at least it is made available to the components that make up our application now that the AuthorQuiz component is wrapped in a provider. So the next step is to modify the AuthorQuiz component to read its state from the Redux store instead of passing it in as a prop. To connect a React component to a Redux store, we use the connect function from React-redux. So I will important that and scroll down to the AuthorQuiz component, and wrap the AuthorQuiz component in a call to the connect function. Connect is a function with two arguments, mapStateToProps and mapDispatchtoProps. The call to connect returns another function, and the argument to that function is our original React component. What I need to do next is to implement these two functions, mapStateToProps and mapDispatchtoProps. Now their names tell you what they need to do. We need to map state to props. So the input to that function is the content of the current store. And we want to return the bits of that store that we need for the AuthorQuiz component, and that will be the turnData and the highlight value. The next function is mapDispatchtoProps. Again, the name of the function is instructive. It's a function from dispatch to another set of props. And this time what we're effectively doing is mapping vents that can come out of the AuthorQuiz components to actions that we want to publish to the Redux store. You can see that the AuthorQuiz currently has two events that it uses to pass data back out, onAnswerSelected and onContinue. So they will be the two props that we have to provide. onAnswerSelected, when an answer is selected, we will dispatch an action to the Redux store. By convention, actions are objects with a property called type. Now I'll call this action answer_selected, and I'll include the answer itself in that action. The other prop is on continue. This is the event that occurs when the user has selected the correct answer to the quiz and then click the continue button, in which case we will dispatch another action, and this is a continue action. Looking at the implementation of the AuthorQuiz component, we can say the props that it's expecting, turnData and highlight, will be provided by mapStateToProps. onAnswerSelected and onContinue will be provided by mapDispatchtoProps.

There aren't any props left to be provided by the parent of the AuthorQuiz component, which means we can get rid of these here, and also we don't need to provide the state, since that is now coming from the Redux store via React-redux. Now that the AuthorQuiz is connected the Redux store, it's time to do a bit of work in that store, get some data into it, and get it handling the actions that we're publishing. The way to specify the starting state for our store is to provide a default value for the state parameter. We'll include the authors in the Redux store, the turnData, which we can set to the result of calling getTurnData and passing in the authors. And the other thing we need is the highlight value, which can start out as an empty string. That's our default value. That solves the problem of getting some data into our store, but we still need to make sure that we're processing the actions that the AuthorQuiz component is generating. Most commonly you will see reducers implemented as a switch statement, but we switch on the type of the action. Each different action type will be handled differently. First up, we have answer selected. This is the action that is dispatched when the user selects an answer. When that happens, we firstly need to calculate if they have selected the correct answer. This is something that previously happened in the onAnswerSelected function, so I can take the code from there. And the correct answer is when the author for the current game has written a book that matches the user's answer which is attached to the action. Having calculated if the answer was correct, we can now update the application state. We'll use object assign to make sure that we're creating a new object. We'll default to the existing state and then override the highlight property. If the user's answer is correct, then the highlight value is correct, otherwise the highlight value is wrong. The other action that AuthorQuiz publishes is continue. When the continue action is processed, we update the current state, resetting the highlight value to the empty string, and generating a new set of turnData by calling getTurnData again. Now I shall add a default case that will handle any actions that are not explicitly handled by a case in our switch statement. And in that case, we can just, once again, return the existing state. If we don't know how to handle the action, then we just don't handle it at all. We no longer need the state variable. We no longer need the onAnswerSelected function because we have the React-redux provider taking care of re-rendering the application. When the state changes, we no longer need the render function. We can simply call ReactDOM. render. The compiler is telling me that resetState is defined but never used. So I guess I don't need that either. And it looks like the application is still working, That's really completed the transition to Redux for the game part of the Author Quiz, but remember we also have the AddAuthorForm. So we'll look at that next. As with the AuthorQuiz component, the first step is to import the connect function from React-redux. Then when exporting the component, we use the connect function. Once again, we need to provide a mapStateToProps and a mapDispatchtoProps. The AddAuthorForm doesn't need to read anything from the Redux

store. So, our mapStateToProps can be an empty function. It does produce some actions though, or at least one action. So we do need a mapDispatchtoProps function. And the event that we would like to map to a Redux action is onAddAuthor. When the form is submitted, we'll dispatch an action of type Add_Author, and that will include the author object. So this is going to happen when the user submits the AddAuthorForm. We'll dispatch the Add_Author action. In our reducer, we can process that action, add the new author into the application state. That's great. The problem we'll have is that the user is still sitting on the add author page. We need a way to make the browser navigate back to the main game interface on the root URL. And to that, we need to import the withRouter function from react-router. And the module for react-router is called react-router-dom. We can wrap our exported component one more time. This time with the withRouter function, and that will make sure that the AddAuthorForm component is provided with a history prop. In mapDispatchtoProps we can add a second argument for the components props. Then access the history prop and use it to navigate back to the root url which the router will pick up and render our game interface. Now we're dispatching the Add_Author action. We need to go back to our reducer and make sure that we're processing that action, which involves adding another case, this time for Add_Author. When Add_Author is dispatched, we will update the application state stored in our store, which will be equal to our existing application state, except that the authors array will be the existing array with the new author concatenated to it. Let's see if that still works. Okay, so, here I am on the main game interface. If I select Add an author, okay, that's interesting. The Add author route is broken. Previously we used an AuthorWrapper withRouter to make the history prop available. We've now move that inside the AddAuthorForm components. In fact, we've done that with the onAddAuthor event handler as well. So, that whole thing can go. In which case the AuthorWrapper is now superfluous and we can just delete that whole thing. Instead of using the AuthorWrapper, we can just use the AddAuthorForm component directly, and that's because it's now a component that's connected to the Redux store. The next problem that we're going to have is that AddAuthorForm is connected to the Redux store, but it's outside of the React-redux provider. So, it's not actually going to be able to access the store. To fix that, I can move the provider from the app component up one level. When I do that, we have a similar situation. The app component now wraps the AuthorQuiz component and doesn't add anything. So we can go. And this is nice. This is an example of the way in which things are being simplified by React-redux. Now our routes can map directly to the components themselves without the wrappers. And let's fix the navigation. So now it's time to try adding a new author and making sure that that still works. Let's add Jane Austen. And she's written some books such as Pride and Prejudice and Emma. And now we can add that author. And as before, then it's just a matter of playing the game until our new author is randomly

selected, thus proving that they have been added to the application successfully. So this is the image that are used for Jane Austen. And in the list of possible answers, we have Pride and Prejudice which is a correct answer for that author. So that is working. The last thing I'd like to demonstrate is a little tool that can help you to debug issues with Redux and React-redux. You can see here that in my developer tools I have a tab called Redux. This is provided by a tool called Redux Dev Tools. Redux Dev Tools is available for Firefox and Chrome, and it provides a wealth of information about what's going on in your Redux application. You can see the current content of your store. You can see the actions that have been processed and what values they have. You can record and replay and just generally get a really good feel for what's going on with your Redux store. To get that working, you need to install the, the relevant extension for your browser, and then make a small change in the application itself. Where we create our Redux store, in addition to providing the reducer, we need to add a second parameter. Now if I go back to the application, it's reloaded and you can see that the Redux Dev Tools are showing a user interface. I can have a look at the State tab, and see the current value of my Redux store as all of the books and authors and the current turnData. The game is showing Shakespeare. And in the turnData we can see that the author is William Shakespeare. As we'd expect, if I select an incorrect answer, the highlight value changes to wrong. Over in the list of actions, we can see that there was an answer selected action. I can see that the answer selected action had the type answer selected and the answer Heart of Darkness. If I select the correct answer, we got another answer selected, this one is Hamlet. Continue button will cause the continue action to fire. All along the state is updating. If I was to switch to Add an author, fill out the form, you could see the add author action and the application state would have a new author in the authors array. So that's a very handy tool for debugging Redux and React-redux issues.

Summary

The model view intent architecture is a useful way to think about designing react applications. React provides the view responsible for rendering the model to a user interface. The model can be a plain JavaScript object or something more sophisticated, like a Redux store. And intents are the user actions that drive changes to the model. A state container is intended to centralize application state and simplify managing changes through that state. It's easy to create a custom state container, or you can take advantage of an existing opensource option like Redux. Redux comes with a substantial ecosystem of add-ons, like Redux Dev Tools, and a variety of middleware. When using React and Redux together, the library React-redux provides some useful

features, but don't underestimate the importance of first understanding React and Redux by themselves before adding the extra complexity of React-redux.

Course author

Liam McLennan

Liam is a technology leader, engineering manager and agile product delivery expert. He helps organizations to develop effective technology strategy, then implement that strategy to achieve their...

Course info

Level	Intermediate
Rating	★★★½★ (276)
My rating	★★★★★
Duration	4h 13m
Released	21 Jun 2018

Share course