React.js: Getting Started

Introduction

Introduction

Hello. We're covering the basics of the React.js library from Facebook in this course. I will be showing you how to build a simple UI for a kids' math game with React. React is a JavaScript library for building user interfaces. Note that it is not a framework, or at least it is not marked as one. React is very small and it does one thing and it does it really well--the UI. Some people like to think about React as the V in MVC. MVC stands for model view controller and React is the library to take control of just a view. This is good to remember. One of the most important concepts to understand about React is the component, which takes in two things. A state objet and a properties object. Then the component uses those objects to render some HTML. Think of it as a function with the state and the properties being the input of the function and the HTML as the HTML as the output of it. State and properties have one important difference. The state can be changed while the properties are all fixed values. Components can only change their state and not their properties. This is a core idea to understand in React and we're going to see some examples of that. When the state changes, the component owning that state triggers a re-render. Let's look at an actual example of a component, a simple one, without any input and with just a simple H1 in a div output. On the left side, the component is written in a special syntax called JSX. JSX allows us to describe our DOM in a syntax very close to the DOM we're used to. It is, however, optional. React can be used without JSX, as you can see on the right side. In fact, React just compiles the JSX you see on the left to the pure JavaScript you see on the right and then just work with that in the browser. What you see here is a JavaScript representation of the DOM, which React calls the virtual DOM. The virtual DOM is what gives React its edge and makes it really fast. Internally, when React re-computes in re-render components, it does it all using the virtual DOM in pure JavaScript, which is way faster than the browser's DOM. After that, React does something really smart. It computes the difference between previous and current virtual DOMs and then writes just that difference to the browser's DOM and by minimizing the browser's DOM operations, React manages to be really fast, probably faster than anything else out there today when it comes to rendering the UI.

Your First Component

Let's do a quick React example to get you familiar with the syntax for components. I will be using Plunker for this example. Plunker is a playground tool where we can test our JavaScript HTML in CSS right there in the browser, no need to install or configure anything. Go ahead and click that launch editor button. You'll see this three tabs interface here by default. Let's hide this popular packages tab and instead show the live preview tab (typing). The first tab is the files we're going to be working with. The second tab is where we're going to edit them, and the third tab will show the results of running all the files. To start working with the React library in Plunker, we need to include it first. Switch over to the React.js site and just copy the script line under downloads and paste that in Plunker. Let's put that line at the end of the document. (typing) To work with JSX, let's rename the script extension to JSX. This will tell React to parse it as JSX. Then we can use it as a JavaScript in our HTML file. We need a starting point, an element to designate as our mount node for React. Let's make that a div with an id of root. This will be the element where React is going to take control. Note that other elements on the page might exist without React, but that root div is what ties

React to the DOM. (typing) We're ready for our first component. Switch over to the script.jsx file. (typing) The syntax to create a React component is react.createClass. This takes in one regular object that can have multiple properties, but the only required property on this object is the render method, which is a regular function that returns JSX. Let's make it return a button. (typing) This is a complete and very simple React component; let's use it. To use a component, we need to give it a name we can reference. So let's put it in a variable. Call that button. (typing) The syntax to mount a component in the browser is react.render, which takes in two arguments, the component to render, which is just button in our case, and the mount node where this component should be rendered and that is our root div, which we can reference using get element by id. Just like that. We have a button and it is rendered through a React component. (typing) Let's add some interactivity to this so-far boring example. Let's make that button increment a counter on every click and display the value of that counter as the button label itself. Sounds good? So the label of that button is going to be a number like 5 and when I click that button, it will change to 6, 7, 8, and so on. Since this is something that needs to be changed through the component, it belongs to the state of the component. We basically need the component to re-render itself every time the counter changes. We can't use a property here because properties are immutable. The syntax to initialize the state object of a component is this get initial state function, which returns an object representing the state. The properties of this object are the elements of the state. For our case we need a counter state, which should start from zero, and don't be like me and forget the comma after this function. (typing) Now to actually use this state, or anything else dynamic within the JSX, we place it inside curly brackets. To reference the state we use this.state. (typing) The this keyword refers to the component itself. We're saying use the counter element of this component state. We can see here how the button got rendered with the value of zero. You can try and change that state to see how the button is rendering the values you change. Perfect! We have a state and we have a button that displays that state. Now we need to change that state when we click the button. So we need to define a click handler in that button. React comes with normalized events that are easy to use. For this case, we need the on-click event and we're going to pass it a function name, which is also to be defined on the component itself. Let's call it handle click and to define it, you just add it to the component core object. Handle click is going to read the current counter out of the state, increment it, and then change the state with a new incremented value. We can use the building component function set state to change the state. The value of the new counter is going to be the value of the current counter incremented by 1. And again, don't forget that comma. Let's see that working. (typing) One more time. We define an event handler for the click method. Every time you click, this function is going to be executed. The function reads the current state, increments it, and then sets the state to the new value of the counter. React takes care of all of the rendering needed after these changes. You don't have to worry about that. Simple and powerful.

Reusable Components

So far we've only seen one component so let's add more. Let's split our one component into two. The button to be just the incrementor and let's add a new results component, which is going to just display the value of the counter. For the result component, we used the same syntax of create class. This is a simple one div component that is going to display the counter value. Now notice that the component does not show up yet, because I have not included it anywhere. I just defined it. Let's include it. Let's create one main component to include all the other components. It will render a simple div and within that div let's render the button component and the result component and let's change our browser node render method to render the new main component instead of the button and look at that. I now have both the button and the result components showing up at the DOM. Since we no longer need the button to display the counter value, let's just make its label as +1. When we click that button, the result div will change and we'll see the counter value changes in places of these X's. Now we have a problem. The counter is a state on the button component and we need to access it in the result component, which is a sibling of the button component. So this is not going to work. To make that state accessible to both components, we need to move it one level up and put it on the parent of both sibling components, which is in our case the main component. We just moved the get initial state call down to main, and we should also move the handle click call because it is what we're using to change the state and it should belong to the new owner of the state. But now, our on-click handler on the button is not going to work anymore because we moved it. So here is what we can do to fix that. We can make the main component tell the button component what to use to handle a click and just pass that into the button component as a priority. I'm going to use a different name here for you to see the mapping. The property name will be local handle click and we're assigning the main handle click function to it. Within the button component I have access to this new property called local handle click, which would allow the button to actually invoke the main handle click function. This is powerful. Basically, when we click that button, it would reach out to the main component and say, hey parent, go ahead and invoke that handle click function for me. So the syntax to access properties of the component is this.props. Remember that. Our result component needs to read the counter value and we can use the same trick to pass in that counter value from the main component as a property to the result component. Let's call that property local counter and it is going to read from this.state.counter. Within the result component we can just read the property as this.props.localCounter and now we can test. Note that from the point view of the result component, the counter is not a state. It is just a value that the main component is passing to the result component and the result component will always print that. The state changes on the main component level and the place where you define the state is an important question to answer when designing your react components. Components are all about reusability, right? So, let's make that button reusable. Basically, let's make a button that can increment any number. So you can use it to create a +1 button and a +5 button, and a +10 button and so on. We need to upgrade this +1 to something dynamic. It will be a property of the component. The amount to increment. Let's call it just increment and pass it a numeric value. In the button, we can use that property as usual with this.props.increment. So now we can reuse this component with different increment values. Let's do a 5, 10, and 100. (typing) The UI rendered four buttons now with different labels, but they're not going to work yet. In fact, all of them would still do a +1 at that point because we did not change this part. There are a few ways to fix that. The simplest is to use a local function in the button component and make that local function call the parent's function with an argument of the increment. So let's upgrade this property call here to a local function call. If I just use this.localClick handler, that means call local function instead, and let's define that function, which will basically re-invoke the passed-in property local click handler, which we got from the main component, but now we can call it with an argument and pass it the buttons increment, which is also another property on the button. The main click handler needs to change to accept an argument and use that argument instead of 1. And of course, once again, I forgot that comma. Check it out: +1, +5, +10, +100, +5, +10, +1.

Summary

This module was a basic introduction to React, the V in MVC. We used Plunker to build a small example project on React. Plunker has React support built in and it makes testing React concepts really easy. A React app is a set of reusable components. Components are like functions. They take input and produce an HTML document object model or DOM for short. The input for components is one of two things. A set of properties you can access inside the component with the props object, and the set of state elements that can be accessed with this.state. State can be changed and every time we change the state of a component, React re-renders it, while the props of a component cannot be changed. In React we write the DOM in JavaScript and we call that JavaScript representation of the DOM, the virtual DOM. Writing HTML in

JavaScript is a lot different than what we're used to, but luckily React has a way to write the virtual DOM in a syntax very close to the HTML syntax we're used to and that is JSX. Once we have the virtual DOM described in JSX, we can pre-transform it before using it on our sites. The syntax to create a React component is react.createClass. This takes one argument, a JavaScript object, and we can define multiple things within that object, but the only required property to be defined there is the render function, which is where we write the DOM. The syntax to mount a React component in our main document is react.render and that takes two arguments. The component to render and an HTML node or element to hold our React rendered markup. React also comes with normalized events that work across all browsers. We've seen the on-click event handler and similarly there are other on-something events like on-change, on-submit, and so on. In the next module, we're going to work with some data using React. We will read that data from the GitHub public API. Here is a preview of the final product. It's a simple cards app. You have a form to add a card using the GitHub username, and that card will show the avatar and the name of that user, and you can add multiple cards like this. The users I'm testing with here on the screen are the top committers to the React project. Our thanks go to them and to the rest of the React team.

Working with Data

Introduction

We've seen simple components, we've worked with multiple components, and we've seen how and when to use state and properties of components, but we haven't really worked with any real data yet. So we're going to be doing exactly that in this module and we'll use the GitHub public API for it. We're going to build a simple GitHub card component that displays information about a GitHub user. We'll need to work in one of the lifecycle hooks of a component, namely the componentDidMount, which is the hook at which the component has a DOM stretcher we can work with. Other lifecycle hooks include things like componentWillMount, componentWillUnmount, and a few others. Once we have a card component, we're going to explore how to render multiple of them and have that driven from an array of GitHub usernames, then we'll add a simple form to add a new card to the same array of cards driving the UI.

Build a Github Card Component

Starting from this template here, a React MT component in the script, and the HTML is the same we've used so far where we have root div, the React library, and our script file. Let's start with a card component. This is the box that will display both the avatar and the name of the user. (typing) So let's make it render a parent div and an image element and the name in an H3 element and let's add a horizontal line after every card. Now to render a card example, we need to add it to the main component, which is what we're rendering to the DOM. That made the name show up and of course, we don't have an image source. Let's actually bring in some real data to see how a card would look like. We can copy some data from the GitHub API directly. If we go to https://api.github.com/users and provide any GitHub username there, we will see their data as JSON. Now let's go ahead and copy the avatar URL and the name and use them in our card. That's a big avatar, so let's force a width of 80 on it. (typing) It looks good. So now that we have an idea of how a card is going to look like, let's make it work with the API directly. Ideally, we want to pass the card component login property and have it fetch the information for that GitHub username from the API and to do that, let's initialize the component with an empty state object so that we can change the state later on. What we need here is the right time to make the component fetch the data and that is provided by React through what they call a lifecycle hook. There are a few of them around the lifecycle of a component and the hook we need for this case is right after the component gets mounted in the DOM, since at that point we have access to its structure. Before we add the lifecycle hook, we're going to need to fetch data with AJAX so let's bring in jQuery so that we can use its API to fetch data. Just add it through the popular packages tab. (typing) Back to React. The lifecycle hook we need here is called componentDidMount, which will be a function that would just do an AJAX call to the API. Let's grab that endpoint and paste it for the $.get method from jQuery. In the callback function for that $.get method we will have access to the return object as the variable data and we're going to go ahead and set that variable as the new state for this component. To access the set state method in a callback function like this, we need to declare the component as a variable before the call. Otherwise if we try to access the component with the this keyword within the callback, it will be something else. So now we can do component.setState and pass it in the data coming back from the API, which represents a single GitHub user object. Now in the render method, we can access the properties of this new state, which maps to a GitHub user now. So we can do this.state.name, this.state.avatarURL. So now we have a card component capable of fetching the data directly from GitHub. The API endpoint is still hard coded for Pete Hunt though, so if we add another card with a different login property, we're going to get another Pete here, but the fix is simple. The API endpoint should be constructed using the login property of the component, and now we can say this card component is done.

Taking Input from the User

We have a card component ready. We need to manage a state of multiple cards next. What we have so far is a manual call for two cards. Let's switch this into an array of cards that we can add to. This array belongs to the state of the top level main component, because when we push a new card through that new array, the components need to re-render with the new card. Let's call this array this logins array and initialize it with two logins. Next, we need to dynamically construct our cards based on this new array. Let's create new variable cards to hold this dynamic array of cards. We'll start from the state's login array, and map that into an array of cards, passing in the login info from the logins array. And now we can replace the manual cards we had before with the new variable that holds a cards array driven from the logins array and just like that, we see the new cards rendered in the UI. We made sure this array of logins is working, so now we can initialize it as an empty one and let's finish this app with a form that would add the new card to our array. This would be a new component that renders a simple form with the text input and a button and to render the form in the UI, we add it to our main component, and there you go. We have a form. Let's handle this form's action on submit. Let's create a local function in the forms component, name it handleSubmit. Let's also pass it the event object so that we can trigger a prevent default in that event because otherwise the form will submit normally and refresh the page. To access the value of the text input element, we need to give it a special React reference using this ref attribute. Let's name that reference login. Back in the function, we can use that reference using another special React call. Basically we do react.findDOMNode and pass that the reference we just added to the input. So this.refs.login, which will grab the reference we added below and findDOMNode will make that find the input element associated with it. We can now read and write to that element from here. So we need to add the card and then reset the input back to empty. To add a card, however, we need an action on the top level main component, as that's where we have the array of all cards. So let's add a function there first. (typing) However, we need a way for that form component to invoke this addCard function on the main component and for that we just pass the function into the form component as a property and we can just use the same name here. Now that we have a property we can use to invoke the main components at card, we can call it in this handle submit function and let's go ahead and pass it the login input value here. Since we called the property with an argument value, the main component's addCard function needs to receive an argument as well and use that to add the new card. Let's call that argument loginToAdd. So here's what we need to do in addCard. Grab the current state's login array, push the new login to it, and then set the state to this new array that we just pushed an item to. This should do. Let's test. Add one card. And another. And another.

Summary

We've built assembled GitHub card component that takes in a GitHub login as input and renders information about that GitHub user. We've seen how to fetch the data for that component from within the componentDidMount lifecycle hook. Then we explored how starting from an array of GitHub usernames we can map that into an array of card components and React will just render the multiple cards. Then we worked with a simple form and explored how to access an element in the DOM from within React using React's special refs object and the findDOMNode method. It is really important that you understand the concept of passing properties from the parent to the child and those properties can be regular values or they can be full functions that the children can invoke up the chain on the parent component. Here's a preview of the game we'll be building in the next module. It's a simple math game for kids. The player gets a random number of stars and they need to select numbers that would sum up to that number of stars and keep doing that until all the numbers are used.

Building the Game Interface

Introduction

We're going to be building the interface of our game in this module. It will be mostly markup in CSS, but we're gonna be doing the markup directly through React and we will also render dynamic sections with React collections. We're going to be using Bootstrap for the look and feel of the UI and we will build everything using Plunker. Over the next few modules we're gonna build an in-browser math game for kids. I'm calling it Play Nine. Here's a preview of it. When the game starts, you get a random number of stars between 1 and 9 and you have the set of numbers in the bottom frame that you can use. You can select one or more numbers that would sum up to the value of the random stars. The objective is to use all the numbers in the bottom frame. If you end up with a number of stars that has no possible correct combination then you get to redraw and you can do that five times. After that, if you still end up with a number of stars that has no possible correct combination out of all the remaining numbers, then you lose the game. Let me show you how to play. We have nine stars here so we can do either 9, 8 + 1, 7 + 2, or let's go with the 6 + 3. Check the answer. We get an indication that the answer is correct and we can click again to accept the answer. For three starts, let me first show you how a wrong answer would look like and I can go ahead and undo the number selection and select something else. And now for two stars there is no possible solution among the numbers remaining so now we can redraw, and my bad luck got me two stars again so let's redraw one more time. Now for five stars there is an answer, for three there isn't, redraw. Seven is possible. Two isn't. Redraw, and I'm going to just redraw one more time to lose the game. There you go. Six isn't possible among the remaining numbers. I don't have any redraws left. Game is over. Let me try to win this game real quick. Click play again and play it smart. (typing) And there you go.

The Main Game Component

Let's build our game. So far we've been using the built-in JSX feature in Plunker. You can quickly start a React project in Plunker using this menu under the new button. Select React and you will get ready React template with an external JSX script, which is cool. This is, however, not the current latest React files, so if you want to use the latest, we'll have to do a few slight modifications to this template. We'll need to replace the script line inserted by Plunker here with the latest from React, which we can get directly from the React site. This is the main React library. It does not have a JSX transformer. If we need to use the latest JSX transformer, we need to include it here as well. And now to use it, we include the JSX directly in here, and mark it as type text/jsx and now we're back to a working state for the template, but now through the latest from React. Let's clean up this template. I don't need the read-me file, and let's replace example with an ID of container, and I'm going to also change the message to just hello React and I'm gonna use Bootstrap CSS framework for this project, so let's go ahead and add it to the resources and we can use the packages tab here, and let's also add jQuery because Bootstrap depends on it. And a Bootstrap document needs a class equal container div. Let's give that to our main container here. One cool thing you can do here with Plunker is that you can save your plunks and if you log in to Plunker through GitHub, you can always see the list of plunks you save, which is handy. I'm gonna start saving my work here. You put a description in the plunk and click Save. Let's go ahead and start building our component. We need a main component that will contain all the other components. Let's call it game and make it render a div with an id of game and let's render the header of the game there. To render this game component, let's replace the Hello React h1 with the game component call. And there you go. It's now getting rendered.

The Stars, Button, and Answer Frames

Let's create our first sub component for this game. The stars frame. This is where we're going to render the random number of stars, but for now, let's just build the markup and make sure things look good. So this is going to return a div. Let's put a placeholder dots in there and go include the component under the game component. Make sure the dots get rendered. Now copy this stars component and let's create two new components, one for the button that's going to show up in the middle, and let's call that button frame, and another for the answer box which I'm gonna call answer frame. Let's render these two new components in the main game component and make sure they show up. Nice. Let's start adding the markup for these components. Let's give this stars frame an id and put a well inside it and inside that well let's include a single star to begin with and we can use Bootstrap glyphicons here. The star icon is glyphicon-star. The UI is not updating though so that's an indicator that we have a problem and we do. We're using class here and we should be using class name. This is one of the JSX requirements that you need to remember. We now have a star. Let's add a few stars. For the button, let's also give it an id of button frame and add a primary button with the equal sign in the content. And for the answer frame, an id of answer frame and the well inside that. The markup for the top part of our game is in good shape now. Let's style it a little bit. The stars are too small and too close to each other. I'm going to give them a little bit of a margin and increase their font size. Since we want this top part to be three sections on the same block, let's give both the stars frame and the answers frame a width of 40%, make them float left, and that leaves the button frame a width of 20%. Now let's also make it float left and line it centered in the div. Let's give the content of both the stars frame and the answer frame a fixed height, (typing) and I'm gonna make that button a large button, and let's also give it a margin top and let's add a horizontal line under the header and let's wrap the floating div with a clear fixed div. Looks good. No worries if you don't understand the styling I just did. This course is not about that, but styling skills are helpful. There are plenty of courses you can watch on both Bootstrap and CSS itself.

Random Number of Stars

Let's make our stars frame render the actual number of stars as it should do in this game. For a starter, let's make it render the stars from a variable. I'm going to call this variable number of stars and start with a value of 5. So to render a dynamic number of stars from a variable like that, we need to iterate over that variable's value and for every iteration, add a star. There are a few ways to do that. The simplest is just to use a for loop. From zero to our variable, and inside that loop, push one star to our stars array. In the components render method, we can just use our stars array directly in curly braces like this. When React sees this array, it will just join it and render it correctly. Even if the array is an array of values like numbers, React will just render every number in an HTML span, which is pretty cool. Now that we're rendering the stars from a variable number, we can go ahead and switch the test value with an actual random number and we can use math.random here, which gives us a random number between 0 and 1. So if we multiply that by 9, that would make it between 0 and 9, it would never really produce a 9. The max here is 8.9999, but since we shouldn't have a zero in the result anyway, we can just take the floor of this value, which would make it between 0 and 8 and add 1 to that. So this is the math.random expression we need to produce a random value between 1 and 9. We can test this value by changing a small thing in the script to force a refresh and we see that we're getting different random numbers of stars in the UI.

The Numbers Frame

Let's finish the UI markup. We need the frame where we're gonna display the numbers that the player can select. Let's call it numbers frame and let's include it here under everything else. I'm gonna clone the answers frame and rename it to numbers frame, change the id's to numbers frame as well. The div will show numbers. Let's create a placeholder number here. Give it a class of number and a test value and let's add a few more. Now let's go ahead and style them a little bit too. What I wanna do for the numbers is make them into circles and I'll show you the CSS trick for that. Let's select all the numbers divs, give them an inline block display, which would make them render inline, but still have their own box model. Let's space them apart a little bit with some margin. Give them a different background color, a fixed width, align the text center, give them a fixed font size, and now to simply transfer that box into a circle, you just give it a big border radius. 50% should do the trick here. Looks good. So now let's go ahead and render all the numbers that the player can use, which are just 1 through 9. But instead of hard-coding that, let's also use a loop. From 1 to 9, push that markup into an array, use the value of I in the braces, and use the array in the markup, and now we have nine numbers showing up. One last thing I wanna do for the markup is make the mouse hover over effect on those numbers to indicate clickability. You get that out of the box when you use an anchor element in HTML, but if we want to make those divs clickable directly, we should change the cursor in CSS. The syntax is cursor pointer and now if you hover over the number divs, the cursor changes shape to indicate a clickable element. So far, so good. We're done with the markup here. The next module, we're gonna be writing the bulk of the behavior of the game. What happens when you click the number and how you can get an answer right or wrong and a few other features.

Summary

We finished the markup of the game in this module with the help of Bootstrap and some CSS. We've seen how to either use the built-in JSX feature in Plunker or include the newest and latest from React. We explored how to render a random number of stars in the game's top left frame. The button frame and the answer frame are both just static markup so far and in the bottom frame we rendered the nine numbers with a loop and made them show up in clickable circles. Now it's time to add the interactive actions of the game, which is what we will be doing next.

Numbers Selection

Introduction

We're going to implement the number selection in this module, which is the core of how I use the word start playing the game. They are going to see the random number of stars and select one or more numbers that would sum up to that number of stars and they will do that by clicking on the circles in the bottom frame. When a player selects a number, we want that number to show up in the answers frame and we want it to be disabled in the numbers frame so that the player can select it again. When they submit or accept an answer, we want to reset the answers frame. We will be managing a state of selected numbers for that and we're going to write a function to add a number to that state and also remove a number from that state. This state element is going to be a regular array and resetting that is as easy as re-initializing it with an empty array.

The State of Selected Numbers

I resized the tabs here to give the middle tab a bigger space and the UI of the game did not like that, so I'm gonna change the CSS a bit to make the game render better on a smaller screen. (typing) Alright, so the first thing we need for this game is the state structure. Let's make the main game component initialize as a state and the first state property I want to add is the selected numbers array. I am going to use this array whenever the player clicks on a number in the numbers frame to select it and let's seed this state with a test array and consider the numbers 3 and 6 both selected. This state needs to be used by two components. The answer frame components need to display all the selected numbers and the numbers frame component needs to disable a selected element so that the player cannot select it one more time. For a component to access the state of its parent component, the parent needs to pass the state into the child component as a property. Let's use the same name here for the property. Inside answer frame, we now have a property called selected numbers and it will have the value of the game's selected numbers state. The syntax is simple. This.provs. the name of the property we passed it. Remember, this is an array and we start it with 3 and 6 and React will just render that array in the DOM and give each number its own span tag. To make these numbers in the answer frame look like the numbers in the numbers frame, we can just apply the same style to them. They are under the answer frame.well span, and there you go. We have circles here too and just like that, we passed the selected number state down to the answer frame. We also need to pass it to the numbers frame. We can use the same syntax for that. Inside the numbers frame, this is what we wanna do while we're rendering the numbers 1 through 9 in the loop, if the number to render is selected, which means if it is within the selected numbers array, we want to render it differently. Let's give it a different class name. I'm gonna create a class name variable and within the for loop, add a new selected dash class, which is going to be either selected-true or selected-false. So we append to that string the condition that determines if the number is selected or not and for that we can use the index off method on selected numbers. If the index of the current number to render within the selected numbers array that's coming through the prompts of the component is greater than or equal to zero. That means the number is a selected number, so that number would have a selected-true class, while the non-selected numbers would have a selected-false class and we need to use this new class name variable down on the number marker. That should do the trick. So now we can go ahead and style the selected numbers differently and the numbers frame, if the element has a class of selected-true, let's go ahead and give it a lighter background color and also a lighter font color. So that looks good for a selected number. One other thing we can add here since this element is selected, it shouldn't be applicable. So we should change the cursor to something like not allowed, which would show up like this in the UI. Pretty cool. (typing)

Selecting a Number

Now that we implement the selected numbers UI changes, we can go ahead and get rid of the test value. Selected numbers would start out as an empty array. So now let's create a function to select a number. I'm gonna call it click number. This needs to be on the top level game component, as it will work with the selected numbers state; however, we need to invoke it on the numbers frame component. So we need to pass it as a property to that component. I'm gonna assume that this function will receive the clicked number as an argument. The clicked number implementation is straightforward. It will change the state of selected numbers and add the newly clicked number to that state and we can use concat here to combine the new click number with the current array of selected numbers and that's it. Now inside the numbers frame component we have access to this click number function through the props of the component. Let's put it in a variable and what we wanna do here is call the on-click for each number and pass it in the current number as an argument for that function call, but we can't do it directly like this because that would execute the function at runtime. What we need here is a closure of the variable I that we can define in here using the bind method, which would basically create a copy of this function that would remember the value bound to it. So when the player actually clicks a certain number, it will have a copy of the function that remembers the value of the clicked number. This is a bit of advanced JavaScript here and if you don't completely understand it, you should probably read a little bit on closures and bind method, but it's a trick that's commonly used in React apps. So now we can test. Click a number to select it. It shows up in the answer frame and gets selected in the numbers frame. While the UI for this looks good so far, we haven't really implemented the condition around selecting a number. I can still click a selected number and reselect it. So we need to block this and we can do that using an if statement around the setState call inside the click number. We can only select a number if it's not selected already, which translated to JavaScript means if the index off this click number within the selected numbers array is less than zero. That means it's not selected and you can go ahead and select it by changing the state. That should do. Let's test. Select a number, and we cannot reselect it. There is a small problem so far though. Whenever we click a number, the number of stars refreshes and gets a new random value. That should not happen when we click a number. It is happening because we're generating the random value inside the render method of stars frame, which triggers every time there is a re-render in the game component. So what we can do here is upgrade this random value into the state of the game component itself. So let's create a number of stars state, initialize it with the same random expression we used, and now pass it into the stars frame as a property and just use it directly in the stars frame component. Now if we test, we can select a number and the number of stars does not change. So far, so good.

Changing The Answer

How about we implement a way for the players to change their answer? Let's say they click the wrong number or they tried a combination that's not correct and they wanna fix it. So let's make them roll back a selected number if a player clicks on the number in the answer frame. Since this is also a click number action, the click number function name here is going to be a bit confusing. I'm gonna select all the places where we used click number and rename them all to select number. Names are really important and they're sometimes hard to get right. Now that we have select number, let's write an unselect number function. Similarly, let's make this function receive an argument of clicked number. To remove an element from an array of elements, which is the selected numbers state in this case, we can use the splice method in JavaScript. We first need to figure out the index of the element to be removed in the array, and we can read that with the index off and then the splice method syntax to remove one element is, first argument is the index of the element and second argument is the number of elements to remove, which is just one in this case. And then we need to update the state of selected numbers with a newly spliced array. To be able to invoke this unselect number function from within the answer frame component, we need to pass it down as a property for the component. Using the same syntax here, and inside the answer frame component, we need to make a small change first. We are rendering the numbers directly from the array of selected numbers, which just automatically gives every number a span, but we now need to define a behavior on every number. The unselect behavior. So what we need here is to not render the array directly and instead create a full array that has elements we can add behavior to. Since our starting source here is an array, we can use the JavaScript map function to transform it into spans of HTML. Within the map, whatever we return will be used to transform every element in the original array. The result is what we can use inside the markup of the component. So far, this is equivalent to what we had before, but now we have a way to further customize our markup, which is the goal here. We need an on-click handler on every span, and the function to invoke is the unselect number property of this component, and once again, this needs to be a closure on the variable using the bind method, and now we can test. Select numbers. Now we can unselect numbers. The game is starting to shape up now.

Enhancing the UI

I'm gonna do a little bit of refactoring here. We're reading selected numbers out of the state here multiple times. I'm gonna reduce that into just a variable and use that variable instead. And actually do that for the number of stars as well. One enhancement I'd like to do is to disable the check answer button if you don't have any selected numbers and for that, the button frame needs access to the state's selected numbers, so let's pass it through. In the button frame, let's create a variable disabled and the condition for that is easy. It's basically if the selected numbers property that we just passed into the component does not have any elements and we can go ahead and pass the value into the HTML disable property and React is smart here. It will just not include the disabled property if the condition is false, which is what you'd expect, and right away the button got rendered as disabled and it gets enabled when we select a number. So now the features for playing the game are in good shape. What we need next is to implement a way to check and accept the correct answer and reach a winning or losing state. This is what we're gonna be doing in the next module.

Summary

We've made good progress on implementing the first few functions for a player to play the game. They can now select an answer and that would update the visuals in the UI to represent that selection and disable the selected numbers. Players can also change their answers just by clicking on those selected numbers in the answers frame. We've also made a few enhancements to the code in the UI. For example, if we don't have any selected answers, then the check answer button has nothing to do so we disabled it for that state. In the next module, we're gonna finish this game. The features we need are to verify an answer, accept an answer, and add states to represent winning or losing the game.

Game State

Introduction

We're going to finish the MVP version of the game in this module. MVP is the Minimal Viable Product, which is basically your app core features that can be used by earlier adopters and would allow them to provide practical feedback for you next iteration. So to make this game usable, we need few things. We need a way for the users to check an answer then accept an answer and figure out if there are no more possible solutions left. We're going to also implement a way for the player to redraw the numbers of random stars. And we'll see how many redraws we should give the players. The final frame we want to add to the game is the one that would display the done status and give the players a way to play again.

Verifying an Answer

So far, we have implemented a way for the player to select an answer and they can also change the answer. Now is the time to check the answer which is what will happen when the player clicks that big equal sign button. For an answer to be correct the sum of all selected numbers should equal to the current numbers of random stars. Both of these values depend on the Main State. So lets create a check answer function in the game component itself and inside that function we're going to create a correct variable and the condition for a correct answer as we described it is when the current number of stars equal to the sum of selected numbers. We don't have that variable, I'm just using what reads good here but maybe we should make this sum of selected numbers itself a function of the component, prefix it with a This Call and add parenthesis to execute it. And now lets go ahead and implement this new function. Starting from the original array of selected numbers that we have in the component, state we can reduce that into the sum value with this reduced expression and let's give this reduced call a starting value of zero in case the array was empty. Now back to our check answer function, we calculated the correct state of the answer and to keep things simple here, I'm going to go ahead and save this correct variable in the state as well and I will initialize this new correct state with a Null value, with Null here meaning there is no answer being checked for correctness yet. Next, lets pass this correct state down to the Button Frame using a variable here and reading the state into that variable. Inside the Button Frame, depending on this new correct state we need to do a few things. Let's read the property into a variable first, and then let's use a Switch Statement over this variable and we have three cases here, when correct is true and false. Now let's just put Brake statements here from now, and then the third default case is our regular check answer button that we have so far. Let's move that into this new Button variable and also the Disabled Flag should be moved too. And let's use the Button Variable inside the component marker. We know things are working so far because the button is rendering here. So what should we do when there is a correct answer? Let's put up another button up there, change it into a Button Success. And it's not going to be disabled and for the content of the button, let's put a glyph icon of K in there. For the false case lets do something similar. Make it a button Danger and a glyph icon Remove The last piece of the puzzle here is to make the blue equal sign button invoke the check answer method that we wrote. And for that we need to pass it into the button frame like everything else and then on the button marker wire the check answer property into an onclick handler, like this. We can now test. Select the two, check answer, correct. One more time, select one, check answer, wrong. This is good to go.

Accepting an Answer

Now we have a way to verify an answer. The game goes on though after that. So if we have eight stars, select an eight, check that answer, correct. We can still select other numbers. What we should do here is if the player selects a number is to reset the correct state, and we can do that inside the Select Number function. Set correct to Null, and we should also do the same inside the Unselect number function. Let's test that, wrong answer. You can change the answer and the correct state resets, and even on the correct answer, you can still choose something else. Let's now create a way for the players to accept an answer. This will be another top level function on the game component, but for such feature to exist we need to introduce a new state because the action of accepting an answer should mark the accepted numbers as used and that's what will eventually determine if the game ends successfully or if the player should get a game over. Let's initialize Used Numbers array on the state. Start it out with a four and seven for testing and down in the render method, let's read it into a variable, and pass that variable into the numbers frame component. Now inside the numbers frame component, we're going to add another class to every element. It would be used dash, true or false based on whether the number is within the used numbers array or not. And of course we need to read that variable first. This should do. Let's switch over to our style sheet and try to style the used numbers differently. Those are successfully used, so I'm going to give them a shade of green. Using RGB math here, let's make the background a light shade of green, and the font color a darker shade of green. So that looks good for a used number, let's move on. Now that we know the UI will render used numbers differently, we can reset the test and start the used numbers state with an empty array, and now we are ready to implement the accept answer function. The first thing it will do is change the used numbers and add to them the currently selected numbers, which we can read from the state. And after that we need to update the state. Once we accept a number we need to reset the selected number back to an empty array, and we need to update the state's used numbers with the value we calculated. We should also reset the correct state and at that point, the game is ready for a new random number of stars. So let's use the same expression to give the number of stars a new random value. Testing, and it's not working because we did not wire the button to invoke the accept answer yet, so let's do. The button frame needs access to this Accept Answer function so we should pass it in as a property. And within the button frame the successful K button needs an onclick handler within which we're going to invoke the accept answer function that we now have on the props of the component. Let's test now. Select a five and hit the button, one more time to accept and now the five is marked used. Select one more, accept, used.

Redraws

We have a way to check and accept an answer now. Let's try to win this game. Select good answers and accept them. Keep going, keep going. We're to game over pretty quickly. There are no possible answers in this case of four stars. It is pretty hard to win this game as is, so I'm going to introduce a feature of redrawing the number of stars. Let's add a button under the main button and inside that button let's put a span of glyph icon refresh. Put some line breaks between the buttons and make it warning for the orange color. And let's also make it access button. So what should happen when we click this button? Let's wish there was a property on this component that we can call with the name of redraw and let's go up the chain and pass that property, assigning it to a function on the main component and now we can define that function. All it needs to do is change the state and generate a new random number of the the number of stars state element. It should also reset both the selected numbers and the correct state if we have any, and now we can test. Select a number, hit redraw, and get a new random value for the number of stars. All right, now that we have this feature let's try to win this game one more time. Select all correct answers, keep going. Keep going. And we get to this state where it doesn't really have any answers, so now we can redraw. So six we can do, check accept. Two we can't do, redraw and keep redrawing until we hit a five and there you go. We have all selected numbers now, So the game is now too easy. You can just keep redrawing and you'll eventually win. So we need to balance between no redraws and too many redraws. I'm going to test a limit of five on the redraws. So lets implement that. This would be something that we need to keep in the state. Let's call it redraws and I want to display it right there on the same button. Let's go ahead and store it in a variable in the render method, pass it down to the button frame as a property. Within the button frame, we can display it on the new redraw button by reading it directly from the props object, but of course that should go into curly braces. And there you go. We see the number five on the button now. It is too close to the icon though so I'm going to stick an unbreaking space entity in there, all right, looks good. But it doesn't work yet, what we need to do to make it work when we click that redraw and invoke the redraw function we need to decrement the redraws straight forward. So inside the redraw function when we're updating the state. We should also update redraws and set that to the current redraws minus one. Now we can test and it works. It won't stop the action though if we have zero redraws, we should not be able to redraw anymore. So let's write an If statement in the redraw function to guard against that. Only do this redraw action if you have more than zero redraws in your state. And now if we test, zero redraws is where the button stops.

The Done Status

There are a couple of enhancements that we can do here before we proceed. This redraw button should be disabled when it reaches zero. The player shouldn't be able to click it. Implementing this is easy in the button frame, and on the redraw button element we can add disabled property that gives a true value when the redraws are zero. And if we test now when we have zero redraws, we can not click that button. Another enhancement we can do here is the math random expression. We've already used this expression in a few places in the code, and that's duplication of logic there. What we should do here is extract that expression into a new function. Let's call that function random number which will just return the exact same expression. And now we can use this function to replace all the usage of random expression that we had before. Now it's time to implement an end to this game, which is going to be one of two things: either a game over or a successful run where all the numbers get selected. Let's manage this game state with a new element on the state. Let's call it done status and for testing we'll seed it with a 'game over' message. For the display of this done status, let's create a new done frame component which will render div. Let's give it a class of Well and text center and inside that div let's just render a message in an H2 with dots place holder for now. The UI is not updating though, which means we have a problem and right here I forgot to put a comma before the new element on the status object, and now the UI is back in render. But we haven't really included the done frame yet, so let's add it right under the numbers frame. We see the dots now and for this frame to display the actual done status message we need to pass it as a property and read the variable we used out of the statement. In the done frame component now, we can read the done status from the props object directly and now we see the status. So if we have a game over or even if we have a successful run, basically if we have any done status, we shouldn't really display the numbers frame, we should just display the done frame. And if we don't have a done status we shouldn't display the done frame, so it is going to be one or the other based on the done status value. Let's create a new variable to hold either the number frame or the done frame. Let's call it bottom frame and the condition to control that will be just the done status itself. If we have it, render the done frame. Otherwise, render the numbers frame and in the mark up use that new bottom frame variable and now in the UI we only see the done frame, and if we reset the done status to null, we're back to seeing the numbers frame now.

Winning and Losing the Game

Now that we have a done status and we're displaying it whenever it changes, we need a way to update it according to the game rules. Let's create an update done status function on the game component. Inside there we'll need two conditions: one for a successful run and one for the game over case. The successful run condition is easy. If at any moment we have exactly nine used numbers, we won. Let's update the done status in that case to indicate so and after that we should return from this function. If that was not true then we should check if the game is over, and the game is over, if there are no more possible solutions, assuming we have such a function, I'm going to just use it here and for that case, we're going to update the done status with a game over message. We should only do this whole check though if we don't have any redraws left, cause if we do have redraws left then it doesn't matter if we don't have any possible solutions. We can still redraw and continue. All right, let's now implement this possible solution function. It will need two things to determine its value: the current number of stars, which we can read from the state, and the list of possible numbers which we don't have but can calculate from the used numbers. Let's initialize that with an empty array and also put the used numbers in a local variable here. The easiest way to calculate the possible numbers is to loop through one to nine and exclude the used numbers only, pushing everything else in the possible numbers array, so now we have a list of possible numbers in an array and we have a value of number of stars and what we want to answer here is the question, does the array of possible numbers have any combinations of numbers that sum up to equal the value of the stars? This is a popular array logic problem which has some obvious cases but the general approach is to calculate all the combinations in the array and loop over them, checking their sums against the value of the stars. The exact solution to this problem is a bit involved and is probably beyond the scope of this course, so I did prepare a version of it, and I'm just going to use it and I did save it in a Just For You under this link. So you can just go ahead and copy the content of this into you script, read over this function and try to understand what it's doing. There are few advanced techniques here, but it's always fun to decipher, so now that we have this update status function, we need to invoke it, and we should do that in two places, whenever we accept an answer and whenever we update the redraws. These actions can lead to either game over or game success although to be exact the letter is only for a game over. We need to update the done status though after we're done updating, everything else in the state because the update done status function depends on the component state itself but simply calling the update done status after the set state function call is not going to work because the set state call is an asynchronous call by nature, React gives us a second argument to Set State which is a call back function we can use to fire any command when React is done updating the state. So let's use that to invoke the update done status function and we need to do that in both accept answer and in redraw. All right, let's try to get to a no possible solutions state accepting the answers we could, we should also drain our redraws first. Keep going, keep going, keep going. And now I have this five, which is a case of no possible solution. But the UI is not updating, which means we have a problem in which case, the console is your friend and in our console I can see this message, this dot possible solutions is not a function, so we used our function in plural and singular form in two places. Let's just keep the singular version and let's try to get to a game over now, drain all the redraws, select the correct answer when we can, keep going, keep going and there you go. We hit a no possible solution state which updated our done status to 'game over'. At this point there is nothing we can do in the UI. So we should give the player a way to reset the whole game and play again. Let's add a button under the done status message. Call it play again and give it a button default class. And let's update the done status for testing of this button, and let's prepare the function this button is going to call. Name it 'reset game' and inside that function, what we want to do is replace the whole state of the component with whatever we can read from the initial state call, the difference between replace state and set state is that replace state just drops the whole current state and replaces it with whatever you supply, while set state does merge what you supply it with the current state. We have a reset game function now which we need to pass through to the done frame and inside the done frame we can make that button invoke the reset game with a simple onclick handler so testing that now. Redraw, clicking 'play again' reset everything. Let's test the game over state one more time. Drain all the redraws and try to reach a no possible solution state. There you go. And now we can click 'play again' and it works. So now let's try to win this game, select all good answers, keep going, keep going, keep going. Redraw when you need it and look at that. We're lucky, a win with four redraws left, done. This is the end of this course, I hope you enjoyed it and have fun playing the game you just wrote with React.

Summary

We finished the most important features of the game in this module. We started by implementing a way for the user to check an answer and accept it afterward. These features allowed us at that point to officially play the game. It was too hard to win the game though, so we introduced a feature of five redraws to balance the level of the game, and last thing we needed for the game is to display whether a user won or lost the game whenever they reached one of these states, but for that we needed to calculate the state of no possible solutions and we used that to figure out the state of winning or losing the game and we displayed that to the user. This course was an introduction to the basics of React. There are lots of great resources you can use now to continue learning React. The official site has great guide articles that you should read. You also have a good collection of videos there. I also recommend that you continue adding features through this game. For example you can introduce a 'play timer' and manage points based on how fast you can win this game and also how many redraws you used. You can also get the player information and when they win store their points in an online database and then create a leader board widget for all the players.