Testing

Video Transcript

Video 1 – Introduction To Create React App

So, up till now, we've been developing our web components in what we call a standalone version of React. But React has what they call a command-line interface that allows us to create a whole project. So, if we type npx create-react-app, it'll produce a project folder with a number of subfolders and a number of skeleton files. The purpose of this is just to standardize the naming convention for folders and the files. But this is what you'll see in industry, and so, you should be exposed to it. So, we'll develop a simple web component in this new style, and you'll see it's quite easy. It's just naming conventions.

Video 2 – Install Create React App

So, React gives us a starting configuration that we can use. To install it, we do 'npx create-react-app'. And then we give it a directory name. So, I'm creating it under my react directory. But we'll give it a name, and we'll have 'cartsoln01', I like, '01'. Okay, so this is going to do an install. And as you see, it's saying, 'Happy hacking!' for us. Let's look back up here. We need to run now, 'npm start', and that'll start a 'development server'. So, it's going to spin up a server that we can hit. So, there's various things here, like 'Bundles the app into static files for production'. So, that's really good. When we want to deploy it, this is exactly what we need. And this, 'Starts the test runner', and removes this tool. Okay, so let's take a look quickly at what it's built. So, it's built this directory structure for us. Here are the node_modules that it's loaded.

Here's some public images that it's using. And here in source src, we've got a number of JS files. One that we're interested in is, we'll look at index.js. So, this just does the 'DOM.render'. And we should look at App.js. So, this is the main starting point. And we'll see here that it's going to load up an image that is in that public directory, the ' "App-logo" '. And here, it's going to just say, 'Learn React'. So, let's see if we can get control. We'll put, 'Let's Learn React'. Okay, so we'll save that. Now, let's go and do that 'npm start'. So, let's go back up to our, and I'm going to 'clear' this. Okay, so let's go into the directory. So, we need to go into 'cartsoln01', and we'll do 'npm start'. So, it's starting a development server. Let me grab that server. Here we are. It's hitting localhost, and yup, we've got control. Let's Learn React.

Okay, so I've bought up here our present page on localhost:3000. Now, let's go back and make a change to this and see how it works. So, 'Let's Shout Hello !', and I'll save that. Notice that it reloaded. So, it's watching our edits, and that automatically reloads this page. So, that's really very nice indeed, that any changes we make, we can immediately test them. So, let's for example,

take out the image source. Save it. Yup, it's gone. This is our development/debug environment if` you like, that the server will automatically reload the page for us and we can iterate fairly quickly. So, that's spinning up Create React App. So now, let's go and see if we can develop a real project in this.

Video 3 – Create React App - Shopping Cart Example

So, we saw how to develop an application using Create React App. And now, let's see if we can put our shopping cart into this structure. So, we note that App.js makes the main call here. So, it's probably in here that we're going to insert our products. Remember we have a product's web component. So, let's go and get that from the file. So, I'm going to go back to the SHOPPING-CART. I'll save this. And in the SHOPPING-CART, let's get our solution. And I'm copying everything. And now, let's go back to 'create react', okay. So, what I'm going to do, and this is pretty standard, is under src, I'm going to create a folder called Components. And under Components, I'm going to create a file, and let's just call it products.jsx. And in here, I'm going to put that file. Now, we'll notice that here we need to 'import' things like 'React'.

So, let's copy that. And here for sure, we know we need to import React. So, at the top, let's 'import React'. Now, I know I'm using ReactBootstrap, so we need to load ReactBootstrap. And that will be, so 'import'. I believe we'll be importing from this into. So, let's, we'll need to load that up using npm as well. And I know that I've got axios, so, and I believe it's called ' 'axios' '. So, those are the main things I need to load. Let's go and load them. So, I need to, I'm going to break out. So, now I need to do 'npm install', 'react-bootstrap', okay. And 'npm install axios'. Now, I'll be able to import them as I did. They'll be there. Okay, so that looks good. Let me 'clear' that, and I'm going to spin up React again. Okay, good. And I'm sure we'll have errors at the moment with this, but let's go back now. And so, both of these now should be okay.

So, let's go down though, and correct some other things here. I need to get rid of this loading products into the renderer. We're going to do it through App. So, I do need to make sure now that I'm importing the right things. So, for example, here, okay, I'm importing all of these. I can do it. Instead of doing it here, I could do it outside. It probably would be here, for example. I could 'import' all of these 'from ReactBootstrap;', but it'll have to be, ' "react-bootstrap";', okay. So, that will import all of these. And I think we're looking good for everything that needs to be imported. Let's save these. So, I need to save App.js to reload. And in here, I just need to load up. I don't need any of this. Let me take over. And so, I'm going to put '< Products >' in here. And I need to make sure I load it. So, I need to 'import'. And I'm importing it here. It's in Components.

So, I need to be a little careful here. I need to import it 'from "./Components/' and '/products";'. Okay. And I need to make sure I export it. So, down here, I need to 'export default' and it's going to be 'Products;'. Okay, let's take a look. Pretty good, pretty good. So, that's brought most of our, let's just take a look. Yet brought everything in. It's working down here. Can we delete? Yes, we can delete. It looks like some things are not working. So, it looks at the moment like we haven't

loaded Bootstrap. So, we need to go back and actually, do an npm install on Bootstrap. And then let's do that first. So, let's control 'C' out here. So, we'll do 'npm install bootstrap'. We'll go back now and kick up React again. And now, we need to go back and add it to 'bootstrap.css' under index.js. And that should do it, let's see. Yes. So, now we're back to having it exactly as we had it before. So, that works. It's doing everything that we expect it to. And it's keeping track of the stock, yep, going down to 2. And then if we delete it, yes, it goes back up to 3. Everything is synchronized. So, we've successfully imported it into the Create React setup. So, we can continue from here.

Video 4 – Build Static Website

So, let's talk about static websites. The advantage of a static website is that it delivers the same content out to everyone. And once it's delivered that content to your browser, there is no more interaction back to the server. So, our standalone, when we've been building our standalone, we've been doing it by running 'http-server'. And you see it here it's running on port 8080. And here, we're using our standalone, and actually, hitting port 8080. So, you see me there reloading the website, and there's just one communication. So, it's downloaded the website. And now, no matter what I'm clicking on here, there's no interaction back to that web server over here on port 8080. So, we've already got a static website with the standalone. Now, we also built with Create React App, and that runs its own web server as well. Let's take a look at that version. So, here we are in our Create React App version.

And to run that, we need to do 'npm start'. And that starts up its own server that happens to run on port 3000. So, here we are on port 3000 now, and we get the same website that we had before. Now, over here, you see that it was interacting. Here we've got the network traffic. And you can see the files that are being downloaded. Now, let's take a look. I'm going to bring that across here. Let's look at any network traffic as I'm hitting buttons. So, submit that. No. There's nothing happening over here. Let's take a look at the elements on the page. So, here is our ' "root" ' '< div >', and now our ' "container" ', and we've got three rows. And these are the different rows. And here's the 'Cart'. And there's a '< div >' at ' "accordion" '. And there's three cards open. These are the three cards. Let's delete. Let's look at this ' "card" '.

And it's got a ' "card-body" '. Now, let me delete this card, and we can watch the interaction here. Now, that one disappeared, that whole card from the ' "accordion" ' disappeared. Now, again, another ' "card" ' disappeared. So, the way React works is sliding in and out of these divs, different content. So, there is only one ' "root" ' '< div>', and we slide in our web components and slide them out. But all that interaction is done on the browser. There is no communication back to the web server. If we look at network traffic, nothing else has happened since the start. So, to package up this React app, what we need to do is we need to actually, let's get out of the server. And we need to do 'npm run build'. And this will build a static website for us. Okay? We see that we've got

a directory called 'build'. If I 'cd' into that, we'll see that we have an 'index.html' file. Let me run an 'http-server' here ourselves.

I'll spin up an 'http-server'. And I want to hit this index file because now it's a static website. So, here we are. We're running on port 8081. So, now I'm going to go to port 8081, and it'll hit the index by default. And here we are, we've served up that website. Now, you can see it's transferring quite a bit of information. But now, we'll see that I can do things on here, and there's nothing happening over here. There's no communication back to that webserver. So, that's the way you build a static website for deployment. And to deploy it, all we need to do is to copy this 'build' directory. So, we'll copy this, and in my next video, let's deploy it to the Cloud. We'll deploy it to an S3 bucket on Amazon Web Services. So, we'll be taking this 'build' and moving up to Amazon Web Services. So, that's the importance of static websites. And they're much better to defend that we don't have this communication back and forth for attackers to leverage. And today, they are some of the most common websites that you will come across.

Video 5 – Introduction To Deployment

So, once we've built our React app, we're going to need to deploy it. To do that, we're going to create what's called a static website. It turns out to be actually quite easy. We'll just do npm build, and if we've created our site with npx create-react-app, it will just build that static website for us. We can then drag and drop it into our preferred Cloud, and it'll just run. We're going to do it into the Amazon Web Services S3 bucket. And we'll have our website up and running in the Cloud in no time at all. So, sometimes life is good, and this is one of those times.

Video 6 – Deploy Static Website On AWS3

So, we've seen how to get our website running on our local machine. But let's suppose we now want to deploy it. So, here I've deployed it to Amazon Web Services, and you can see that it's working quite happily. So, the first thing we need to do is that we need to go and we need to actually run a build. So, by a build, we mean we're going to build a static website. So, this can be hosted without a web server. Normally, websites are hosted with a web server, but this one is going to be static. And so, to do that, we need to do 'npm run build'. It's quite quick in doing this actually. So, there's our 'build'. Now, We need to deploy. Notice that what it's done is created a 'build' directory here. And if we look inside that 'build' directory, you'll see that we have an 'index.html', which is going to be our main HTML file that will load everything that we need.

So, what we need to do to deploy it now is we're going to copy all the files in this under 'build'. And there's a directory called 'static' there as well, by the way. And we need to take that up to wherever we're going to serve it from. Now, I am going to serve it from Amazon. And Amazon have things called S3 buckets. So, you'll need to get a username on Amazon. Usually, when you log in, you'll be on this page here. And you'll need to go to the S3 buckets. Now, the S3 bucket's

a storage. And well, you can choose various configurations for the buckets. I'm just using a standard configuration. So, let's create a bucket. So, let's give it a name, Let's call it cartreact01. And the region is going to be US, Virginia, North Virginia. And we don't need to worry about that. So, let's just go with next.

Now, we don't want to block public access because we need to have it as a website. I need to acknowledge, they're very careful about public buckets because people have put keys in them and lost keys. Okay, so we've got that. Now, what we need to do is we need to create the bucket. So, here we are, cartreact01. Now, we need to make sure that it's public. So, the first thing what we need to do is go to the properties. And we need to turn on static web hosting. So, we go to that, use this bucket as a website, and our index document if you like, is index.html. So, we just set that. So, we save that. Okay? So, if you want to see how to do this, just look up on Google, AWS hosting a static website, and it'll show you how to do what I'm doing now. We need to give this bucket a policy. And here, I've got a sample one. And now, we need to set this to the website that we're at, which is cartreact01.

So, we just need to type in 'cartreact01'. So, we save that. And it says, it warns us that the has bucket public access now, okay. So, we're all set up on this side except now we need to load up to our bucket, it's empty at the moment. We need to load up the files. So, I'm going to go to the file system. And under this, we see we've got a build. I need to copy the files under the build, and that static one there. Now, don't copy the build itself because we need basically this HTML file to be at the top level of the bucket. Okay, so we've got that and we need to upload. And so, they're there, they're very fast uploading actually.

So, I'm going to find out the URL of index.html. So, this is it. I can copy it. I can copy the whole thing. And now, let's go to our browser and type it in. And now, we're hitting, notice cartreact01. So, we've got our website up and running, and it's fully functional, except that restock the products. Notice that we need to expose our Strapi database so that it could actually be hit from the S3 bucket. It's sitting on localhost on a machine, so it doesn't have a full URL. So, we need to fix that. But that's all. Everything else, we've got a static website that's going to be extremely fast and is a pleasure to work with.

Video 7 – Introduction To GraphQL

In the shopping cart exercise, we built the back-end using Strapi and we used Postman to hit that website. Strapi builds an amazing API. And it built it using REST principles. Now, when we get data using REST, it sometimes returns to us extra data that we actually don't need. Now, if it's the data that's fairly small, it doesn't make much difference. But if it's really large, it can make a huge difference in the response time of our website. GraphQL solves that problem. It allows us to use a POST and to use a query in which we specify exactly the data that we want. And so, we get back from our API just the data that we need. So, GraphQL is a back-end tool in some ways, but we wanted you to see it because it affects the front-end that you'll need to make your requests

using a POST and using GraphQL. So, enjoy this. It's an important part of getting data from remote websites and databases.

Video 8 – GraphQL Demo

GraphQL solves a problem with REST APIs. REST APIs are inefficient in the sense that they often return to us data that we don't need. So, GraphQL corrects that and gives us more flexibility and efficiency in our client-server communication. Let's take a look at one of the problems. So, for example, let's suppose we want to get data about the restaurants. And we just need to get the restaurants' names. So, with REST APIs, we do a GET on /restaurants. And it returns to us all the restaurant data that it's got. And this can often be quite voluminous. So, we'll see that. Let's suppose we fetch day to repair the restaurants and all we want to do is list out the names. So, we do an HTTP GET and we get a lot of data back. Let me just show you. So, here I set up restaurants' data in Strapi. And basically, I give the name of the restaurant, the description.

But they also have created_by that's added. And we put a photo and the photo has lots of information about it, about its thumbnail, et cetera. So, we get all this data back, and this is just one restaurant. And all we wanted was the name of the restaurant. So, GraphQL gives us a way of doing that. It allows us to post queries. So, here is the structure of the query that we put in the body of the post. So, we specify the restaurants, but we specify we only want the id and the name. So, we'll see that what we get back is just this, that we get data about the restaurants, and we have the id and the name. Let's take a look at the example running. So, here's the data in Strapi for a couple of restaurants. There's an Id, there's a photo and the name of the restaurant, and we have a description as well. And let's see now we can hit this as restaurants query. Let's do it. So, we'll go to Postman. So, here we are with Postman. So, let's do a GET on the restaurants. So, we'll send that.

And you'll see here the data that comes back. We got "Papa Razzi", "description", "updated_by", "created_at", "photo". And this goes on for over 50 lines, 55 lines, and then there's the next restaurant. So, REST APIs are really useful, but they are inefficient in that, we get data that we don't need. Let's take a look now at a GraphQL. So, here we are with the GraphQL. We're going to do a post and we select that it's a GraphQL query. And in the query, we just specify restaurants and then the id and the name. These are the properties that we want to get back. And now, we send that. And we get back just this, just what we needed, the id and the name. We've only got two restaurants, but we get the exact information we want. And then we can render that in our API. So, GraphQL needs to be enabled on the Datastore. And then we need to set up our queries so that we can post queries like this. But it's very efficient and we get back exactly the information that we need. Okay, so that's GraphQL.

Video 9 – Testing The Front End

Testing is big enough that it's a profession onto itself. Test engineers, quality assurance engineers, quality engineers are all different names for testing engineers. And in this case, we

wanted to simply give you an introduction. As we mentioned, it's a large topic. There's a lot for you to learn, but we will be giving you here some of the basics. How do you render your application in-memory? That is a headless representation of your application and you check for some of the content that should appear in this in-memory version. Just as well, we will take a look at how to interact with this in-memory version.

So, programmatically add content to your application, programmatically click on buttons, and so on. Ultimately, we will also take a look at some of the mocking that you can do. Whenever you're testing the capabilities of what you're working on, you will often have dependencies on a database or other servers that you cannot fully test yourself, so you will want to mock this part and we'll show you some basics on how to do that. So, with that in mind, let's go ahead and jump into testing.

Video 10 – Testing Overview

In the past, software creation was widely accepted as the system development lifecycle model, where you started with requirements. Here, you try to understand what you were going to build. You move to design, your software architecture. Then you move to your code construction, your implementation, then verification, your testing, and you ended up with maintenance and operation. Accompanying this view, were models for testing such as the one you see here, where you tested your small units of code at the beginning. Then you move to design testing, architecture engineering testing, system requirements testing, and ultimately, user requirements testing. And although today, software creation is very different.

The question of testing is still central to software creation. In the market, you will see terms such as quality assurance and quality engineering. And more recently, intelligent testing, where you make use of the Cloud, you make use of Big data, you make use of machine learning to accelerate your testing with automation and Cloud resources. In this course, we will start you with the basics. We will get you familiar with some testing suites, using testing initially and eventually, creating some simple test. If you want to learn more about the suite that we will be using, which is Jest, you can navigate to jestjs.io.

Video 11 – Install Node

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It is so performant that what was originally part of a browser has become one of the most successful servers in the market. It powers a great deal of things beyond serving pages. It is also used for storage engines. It's used in the IoT community and it has a very powerful, productive ecosystem. That we will initially use here to be able to load some of our packages and do our testing. So, go ahead and navigate to the address you see before you nodejs.org, download the version for your operating system, pause the video, and carry that out. Once you are done, open up a terminal window and

you can confirm the installation of node by entering 'node - v'. And if you get the version back, that means you have installed it successfully.

Video 12 – Test Driven Development

Testing software, given its complexity, has been part of creating software for a very long time, well over 40 years. And as you start to write code and your code grows, keeping a handle on what's taking place, keeping some certainty about what's happening within your code is something that you want to have. Because of it, testing suites and testing ecosystems have emerged around the practice of building software. And one of the approaches is to write your software even before you start to write your code, it gives you some sense of where you're going. It gives you some, some feedback even before you start writing your code. Now, as you get started, this might be too much to have a handle on as you're trying to master a lot of the different technologies.

However, as your code mature and time passes this is something you want to come back to. In this approach that you're seeing before you, it's test-driven development, and in this approach, you write your tests even before you start to write your code. And the approach would be to write some tests that would initially would fail. Then you would write some minimal code that would allow you to pass the test. And then you would refactor, you would improve your code and you would continue to do this until you had pretty good code that captured the intent that you were trying to represent in code. So, if we take a look at a number of steps, you would think about the following, you would write a test, you would watch it fail. You would write some more code, initially, you would fail for a few iterations, then you would fix it. It would eventually pass and to the point that you felt good. And then you would add commit and push into your GitHub repository.

When it comes to React applications, we will be using the functionality that is bundled into the great project of Create React App. There is a great deal to learn there. And if you want to learn more about it, you can navigate to the site online and learn about this project. Now, when it comes to the parts of that bundling that we will be using, we will principally be using Jest and the test library. Both of these projects give us a great deal of functionality that we can leverage a great deal of shortcuts and we will use them throughout our lessons. So, if you want to learn more about Jest navigate to Jest. If you want to learn more about the testing library, navigate to that project, or to Create React App and learn, there's a great deal to learn as mentioned. So, with that in mind, let's go ahead and move to the active part and tests.

Video 13 – Testing Using The DOM

Let's start out by creating a simple todo application that we will use for testing. I'm going to go ahead and 'create-react-app' now. And let's call it 'todo'. Now, let's go ahead and move into our 'todo' application and enter 'npm start'. And as you can see there, we have our default page up

and running. Now, let's go ahead and drag that folder into our editor, which will close our current window. So, let's go ahead and run that again. Now, let's go ahead and open up our app. And we'll go ahead and open up App.js. And as you can see here, we have the code for the page that we're viewing. I'm going to go ahead and remove the 'App();'. And I'm going to go ahead and replace it with the todo app. This is a simple todo app that sets in a list of values which will be the todos, then a value to catch that new todo and handles the submit. And you can see here the rest of the code. This is something we have done previously in the course.

It is no different, perhaps a little bit simpler. This is a little trimmer application that doesn't have remove or any of those additional features. So, I'm going to go ahead here and use the style sheet simply to get some better font. This is just a default style sheet that is part of that default page that comes up. And at that point, I'm going to go ahead and save the file, and our page should refresh. And as you can see here, I'm going to go ahead and enter one, two, three, four. And it works as expected. So, let me go ahead and remove the terminal for a minute here, and show you the full application in case you wanted to continue to work with exactly the application that I have. You can see here the JSX, it's the 'TODO', then the 'TodoList', then the 'form' that takes the new value. You can see here the 'handleSubmit' the two-state items that I mentioned. Below here we have the todo item itself, and you can see there how it is writing that out. It's very simple, minimal application. So, the question is how do we get started testing?

Well, we're giving a template here for App.test.js. And if I were to run that, and I'm going to go ahead and open up here a new terminal in the package functionality script that gets created when we create a React application using Create React App is 'npm test'. And as you can see there executing that test. And as expected, this fails because we have different code than the default code that is added when this default application is constructed. So, let's go ahead and take a closer look at what we want to do, and build up this application from the ground up. So, we're going to go ahead and remove the ' '@testing-library' ' because that has everything. And I'm going to go ahead and add the pieces one by one. So, I'm going to go ahead and start here with an 'import * as ReactDOM', and I'm going to get that 'from 'react-dom' '. And at this time we're going to build our testing up from scratch.

So, I'm going to go ahead and put a more meaningful message here, ' 'ToDo' ', I'm going to remove the current code. And I'm going to start off here by creating a div. And this is the div that we will use to be able to add our application into. This is very similar to what we do when we construct a standalone HTML page into which we add a div element, which we tend to name root, and we will do something similar here. However, we're now doing it programmatically. So, I'm going to go ahead and construct first the div. And so, I'm going to go ahead and enter 'document.createElement()'. And then I'm going to go ahead and select a '('div');' because that's what we want to do. Then we're going to go ahead add and use 'ReactDOM' to 'render' our application into it. And so, this is going to be our modular or our functional component that we have added in App.js. And here, I'm going to go ahead and add that tag for '< App/ >'.

And then, we're going to go ahead and give it the 'root'. So, we have done the same thing that we did previously on the HTML pages when we defined our application, and we set a target that was a '('div')' with an ID of 'root'. And here, we've done the same thing. However, we're now doing this programmatically. This is a headless environment and where you don't see a page. However, the same things are being done. Now, let's go ahead and save the page. And this will rerun the test. And as you can see there, now the test is running. So, we have successfully been able to create an element called 'root' into which we are rendering our application. Next, after having render our component, and let's just make some notes here so we remember our steps. Then we're going to use the 'DOM APIs ()'. And very specifically, in this case, '(query selector) to make assertions'.

The functionality or the support that comes bundled into Create React application is part of Jest. And if you want to know more about Jest, you can look at Jest testing by doing a search on Google and learning more about the full capabilities. Here we will be doing some hello world type of assertions, and we'll start off here by using 'expect'. And then we're going to access that document that we have. And we're going to use 'querySelector'. We're going to go ahead and look for the '('h1')' tag. And this, if we find it, is going to have some 'textContent', and we want that '.toBe ()'. And again, this is part of Jest '('TODO');'. And you can see there, the reason we want it to be '('TODO')' is because that's what we have on the document. So, I'm going to go ahead and save this. And as you can see there, that test is still passing. So, we have been able to do quite a bit. We have been able to render our application onto a virtual environment, an headless environment.

We have then been able to dig inside of it and confirm that we have certain features. In this case, we're simply looking for an '('h1')' that has the content of '('TODO')'. Now, we could as well access other elements on the page, and you can see here that I am looking for a label and I need that label to be '('Add todo: ');'. And if we take a look at App.js, to see if, in fact, it has that label. You can see that we have that label here. You can see what that looks like in the UI. So, let's go ahead and save that page. And every time I save it, the test suite will run again, and you can see there that we're still passing. Now, if I were to change this and save it again, you'd see there that we have now an error. And you can see here that it gives you pretty good instructions as to where it found a problem.

And you can see there that it gives you some pretty good explanations, you know, ' "Add todos: " ' 'Expected:', and a, 'Received: "Add todo: " '. So, now if we remove that extra 's' that we added and we save, then we're once again passing. Now, let's add one more for a button. You can see there that I'm looking for a button, and I need the button to be number '#1' ' in this case. Well, if I reload this page here, you can see that before I add any content. And because I've only looked at the page in that definition that we have here. We have not added any content. We should expect that button to be number '1'. And I'll show you here where the button is, and you can see here this part of the code. And so, now we're going to go ahead and run that code here. And you can see there that it's passing. Once again, if we changed it to '#5' '.

You can see there that it would fail, put it back at '#1' ', it'll pass. So, you can see here how you can start to interact with this definition of this form. And you can start to test the functionality, you

can start to see if you're getting the behavior that you are assuming. So, to overview, we created a React app in the DOM. We then made some assertions, and then we went through some of the document capabilities, that is the browser APIs, that is the DOM. And we confirm some of the functionality, some of the content that we had on the page. So, to get started, let's go ahead and stop at that point. Go ahead and replicate what you have seen, add some additional tests, and confirm your understanding.

Video 14 – Testing Using The React Library

In a previous API, we use the DOM to be able to access the content in our application and to test for certain features or certain strings that we were looking for. In this case, we're going to go ahead and now use the testing library to express our test in a more transparent way. And by that, I mean to construct our test in a way that is closer to how users think. And if we can do that, then we can get closer to the user experience, which will make our tests better. So, in this case, we have added functionality that allows us to take a look at the contents of our page in perhaps a more expressive way. So, we'll start off here by creating an instance of the capabilities that are the following, '{getByText,' and 'getByLabelText}', which will be from 'getQueriesForElement'. And that will come, ultimately, from the '(root);' element. We can now write our test in the following manner. We can 'expect(getByText)'.

And that is not the one I wanted to, '(getByText)'. And we can now look for '('TODO')' directly on the page, 'not.tobe'. So, as you can see, this is more compact, but more importantly, it's no longer referring to the elements on the page. And this is closer as mentioned to the way that a user would experience the page itself. Now, I'm going to go ahead and remove the previous one. And I'm going to go ahead and replace them by the new syntax that we have. Next, I'm going to replay. I'm going to do the test for the label. As you can see there, I am getting label text of '('Add todo:')'. And once again, I'm checking for it to not be null. And then the last test is when the one for the button where I am looking for '('Add #1')).not.toBeNull();' So, again, we're not as connected programmatically to this structure of the document.

We are more at a functional level, checking for features. Again, closer to the way a user would experience the page. So, let's go ahead and save our document. And as you can see there, I have an error. And I believe my casing is off here. Go ahead and save again. And as you can see there, we have our tests passing again. Once again, if we add an s here, and save, our tests will fail. And you can see here that it's unable to find the text of 'Add todos:' . And it shows you what it is expecting on the page and what it has not found. So, we're getting more functional type tests as opposed to more programmatic type tests. And the advantage of that being that testers will start to think more like users, which is a better way to check for functionality.

Now, given that our tests will fail, if the text is not found, we can go ahead and simplify our tests further. And I'm going to go ahead and remove the expect, the 'toBeNull();'. And I'm going to go ahead and save. And as you can see, our tests still passed, and if I, where to we, say add an s

here, we would still get a failure. So, now it's your turn. Go ahead and update the tests that you have written. Go ahead and check for additional things on the page. Write some of your own tests and check your functionality. Make sure that your assumptions about the behavior are what, in fact, is taking place.

Video 15 – Testing Render

We can further clean up our code by abstracting some of the functionality, the boilerplate that we are repeating every time. We're going to do that here with a new method called 'render()'. And inside of it, we are going to paste that code. Now, we will call it with the 'render' call. And we will pass our component. So, in this case, that's going to be '(< App/>);'. And that means this will take a component. And we're going to replace 'root' here for, well not 'root', but '< App/>' for 'component'. And then we're going to 'return' the functionality for 'getQueriesForElement'. Now, let's go ahead and save our file, which will run our tests. And as you can see, there our tests are still running, passing.

And now, that we've abstracted that functionality into a render method, as it turns out, this functionality comes from, we can get the same functionality from the React library. So, I'm going to go ahead and change here the 'dom' to 'react' and get '{render}' from it. And so, now we no longer need this function. I'm going to go ahead and save that. And as you can see there, our tests are passing, and we no longer need 'react-dom' as well. So, there we have the same type of tests that we had before. However, our code is much trimmer, cleaner. Whenever you have cleaner, more readable code, it's always better for maintenance and for readability as well. So now, it's your turn. Go ahead and clean up your code and take advantage of this functionality.

Video 16 – Testing Fire Event

So far, we have verified content within the form, we have verified labels, we have verified buttons, and so on. However, we will also like to input data into this form. We would like to fire events at it, say. And so, that is precisely what we can do and that functionality is built-in. And so, we're going to go ahead and use 'fireEvent', as you can see there. I'm going to go ahead and copy this block. We will use some of the calls that are being made. For now, I will just call this one ' 'fireEvent' '. Well, actually let's be more transparent and we're going to say ' ('add items to list', '. And we are going to get a hold of the '('Add todo:');' form by simply catching what is returned by the 'getByLabelText('Add todo:');. And that is paired with a form, as you can see here, that is 'Add todo:' here, the '< label >' and then the '< form >' and the '< input >' as you can see there on the '< form >'.

And so then, once we get a handle on that, then we're going to go ahead and fire an event. Let me go ahead and grab the right event here. And that event is going to 'change', meaning add. We're going to give it the 'input'. And then we are going to go ahead and specify the content that

we're going to pass in by saying 'target:' and then an object, which is '{value:}', then the content that we're going to pass. In this case, I'll simply say here, ' 'wash car' '. And that will be the event that we're going to add. Now, in addition to that, we need to be able to click the button. And we can do so by getting what returns from 'getByText' when we have '('Add #1');', and this is the starting state. And we will add 'fireEvent.click', in this case, on that button that is returning. And so, that will click and submit. And so, once we do that, we want to confirm that that entry was made, 'confirm data'.

And we can do that by checking that that button has now advanced. So, the '('Add #1');' should now be '('Add #2');' . And that ' 'wash car' ' exists within the application. So, we're going to go ahead and enter here '('wash car');'. And so, at this point, I'll go ahead and run, save and run the tests. And as you can see there, we have now two tests that have passed. If we were to change '('wash cars');' and we saved. You could see there that that is no longer the case. What exists on the page is 'wash car' not 'wash cars' plural. So, if we fix that, then we will get our passing tests once again. So now, it's your turn. Go ahead and add more data to the page. Go ahead and confirm that your data is there. You might want to potentially add other buttons and add other functionality and test here the functionality of the 'fireEvent' and the interactivity with the content on the page.

Video 17 – Testing User Event

In our previous lesson, we added additional items to our to-do lists using 'fireEvent'. It turns out that there's an additional part of the '@testing-library' that is more readable, that is easier to see the intent of the test. And I'm going to go ahead and show that now. I'm going to go ahead and hide the console to give us a little bit more real estate. I'm going to go ahead here and show you the library handle that you need to get. And as you can see here, I am referencing the '@testing-library' and then 'user-event'. And I'm going to go ahead and add an additional test block as you can see there. Let me go ahead and add it.

And as you can see there, the message that I am writing is 'userEvent expresses intent better'. And in this case, we're going to use user-events to allow users to add. So, we're thinking more directly here about the person interacting with the form. And so, what I'm going to go ahead and enter next is that first line of text that gets us, or the first instruction that gets us the handle on the 'render' of the '(< App />)'. Then, just as before, I'm going to go ahead and grab the input element. Then the 'button' that we need to click. And here is where we can express an intent better. So, we're going to use 'userEvent.type'. So, that is we have a user that's typing and is entering into the '(input, "Learn Spanish");'.

Much more obvious what's taking place. And here I have 'userEvent.click' again, pretty straightforward. I think the intent comes through in a clear way. And then as before, we can run our tests. And in this case, we're checking to see that Spanish, learning Spanish was added. And that our form counter button has increased by one. So, I'm going to go ahead and run that now.

I'm going to unhide my terminal. I'm going to go ahead and save. And as you can see there now we have three tests that are passing successfully. So now, it's your turn. Go ahead and rewrite your test. Go ahead and experiment with expressing the previous commands that were entered, the previous test in a new way.

Video 18 – Mock A Dependency

Oftentimes, you'll need to test your components, and they will have a dependency on something that is long-running or perhaps has unpredictable results. For those cases, you'll want to focus on what your part is doing, and mock that other dependency. So, here, in this case, before we add mocking to our todo application, I'm going to go ahead and create a Standalone React app simply to do a very trim, very small test. So, I'm going to go ahead and add that now. I'm going to go ahead and create a new application. And so, I will enter 'npx create-react-app'. And now, that that has been created, let's go ahead and move into the folder. And let's go ahead and start our application. Actually, before I do that, let me go ahead and drag and drop the folder. Now, we can go ahead and open up the console and run the application.

And as you can see there we have the default page. Now, let's go ahead and add a component. I'm going to go ahead here and add a new file. I'm going to call it MyComponent.js. And inside of it, I'm going to go ahead and add the code for MyComponent. This will be a very simple component. As you can see here, all it's doing is returning a '< div >' that says, 'this is my component'. I'm going to go ahead and save it. Then I'm going to go ahead and go back to the main page of the application. And I'm going to go ahead and simplify this quite a bit. I'm going to go ahead and remove the default. And I'm going to go ahead and paste code here that simply, as you can see there, adds a tag for that component and nothing else. I can go ahead and remove the logo here since we're not going to use it. I'm going to go ahead and save the file. And we get a failure because I am repeating that. I'm going to go ahead and save one more.

Now, we're missing our component. So, let's go ahead and add that. Let's go ahead and save. And as you can see there, we get a very simple string saying, this is my component. So now, let's go ahead and move to our test. Let's go ahead and remove the current one. Well, let's leave the wrapping. And we will call this one ' 'mocking test' '. And now, before we start writing our tests, what we're going to do is that we're going to mock our component. And we will do that by using 'jest.mock()'. Then we will reference the component. And in this case, that is the '('./MyComponent')'. Then we are going to go ahead and write our callback. We'll use ES6 notation here. Then our component itself that we're going to write directly into this parentheses here. And we will use ES6 notation once again.

Then our component, which is what will be returned. And so, in this case, I'm going to go ahead and enter here a '< div >'. And let me see, I think the autocomplete does not work that context. So, let me go ahead and write the tag. And inside of it, we're simply going to say, 'Hello World'. So, as you can see here, what we are doing is that we are providing an alternative definition for '

'./MyComponent' '. That is, that when the application is brought in and rendered, instead of using the global component that we have, the one that is in the MyComponent file. We will be using this one that we are providing here. So, we're providing a very simple one. In one case, the message was, 'this is my component'. In this case, it will be 'Hello World'. But we're, the point that I want to emphasize is that we're replacing this component here that we write with the one that we wrote previously.

we will do, once we write more involved components, we will do the same approach. And I just wanted to illustrate it here in a simple way. So, let's go ahead and write our test now. We're going to start off by writing our typical render line. And in this case, we will simply be looking at the '{container}'. And then we will take a look at the text. And so, we will enter 'expect(container.textContent).toMatch('Hello World');'. So, let's go ahead and spin up our test. I'm going to go ahead and run it once again here from the terminal. So, I'm going to go ahead and enter 'npm test'. And as you can see, there are tests passed. But we're to change it here, '(Hello Worlds')'. You'll see there that we get a test that fails.

And that the content of that instantiation of that functional component, the 'App' is, in fact, been mocked for ' './MyComponent' '. So, as you can see, this is a pretty useful set of functionality that we'll want to use whenever we have those type of dependencies that could make tests very lengthy or unpredictable, so that we can focus on the functionality that we have, that we control.

Video 19 – Mock ToDo Application API

Let's add mocking function to our todo application. This is the same todo application that we have been using all along, the one that we can add items to, as you can see here. So, let's go ahead and pretend a common thing to do that each of those items are written back to some API. Perhaps that API persist to a database, but some type of functionality that we don't have control of, that we would want to mock. So, let's go ahead and create a new file. And we'll call this an api. So, let's go ahead and give it a name of api.js. And inside of it, let's go ahead and 'export' this module. In this case, we're going to go ahead and 'export' a 'const' called 'api'. And inside of it, or the definition of which is simply going to be an object. And that object is going to have a method called 'createItem'. And I'm going to go ahead and define that function now. It's going to take a parameter, and we'll call that parameter called '(newItem)' using ES6 notation here.

And then the body is going to be simply 'return' a 'Promise'. And that is the 'resolve()' of the 'Promise'. And we're going to go ahead and pass that back, the '(newItem);'. So, the reason I'm putting that in is because many of the code that you will execute will be asynchronous, meaning you will have to deal with this type of call. And in this case, we're going to address that case here with a very simple method called 'api'. Well, yes, a very simple method wrapped in an object that has a property called 'createItem:', and then the function itself. So, this is our simulation here of the 'api'. Let's go ahead and save the file. And now, we want to be able to use this from the form

or from the functional component for the App where we are creating new items. And so, let's go ahead and open up our application.

And let's go ahead and move to the part of the form where that form, that new item is added to our storage, right? In this case, we're simply save it in state. But in this scenario we're walking through, we want to be able to add it to the API. And so, after we prevent default, we're going to go ahead and enter here 'api.createItem'. Then the '(value)' that we're getting from the form. And then we are going to go ahead and write a function, and that function will work off of the promise. So, when the promise is resolved, then at that point, we're going to pass the persistent item here, '(persistedItem)'. Then the rest of it will be defined as what we had before. Go ahead in that, enter into the body. And so, we've now added a dependency here to a component. And in this case, a module that is mimicking what we would have with a potential API that returns a promise that we need to handle. Now, as you can see we already have a reference there to 'api'.

Now, I'm going to go ahead and move to test, and we're going to do the same. So, I'm going to go ahead and add that here. And next, we're going to go ahead and mock that function, mock that functionality. So, let's go ahead and start here. We'll create 'mock', or we'll give it a name of 'mockCreateItem'. Then we will mock the function. And the way in which we will do it is we will provide that method, which will be '(api.createItem)'. And then we're going to use 'jest.fn()'. So, that is the syntax we need to be able to mock the API. The first thing that we're going to do is that we're going to go ahead and add a string.

We're going to go ahead and add this string as a value that we're going to pass in into 'mockCreateItem'. And we need this to be the type of resolve value for the one type of call. So, this is what it will return if it's called. Now, in addition to that, we will create our render of our app as we usually do, and start to define the content that we're going to input into the form. So, these are things we've done before. We can go ahead and get the label by text, we can go ahead and fire an event. And in this case, we will fire, meaning add an item into the list, which will be ' 'wash car' '. And then we can go ahead and click on the button and make that addition to the list. Now, because this is a promise we will have to deal with in a synchronous method. And you can see here that we get the ability to be able to wait.

And this is something that is part of the library as well. And so, this is why it's underscored there, the 'await', I'm going to go ahead and enter 'wait'. And my underlines should go away in a minute. And actually, the reason that is showing up is because, my function as soon as I entered 'await', is now needs to be asynchronous, and that's to be expected. That's JavaScript, not part of the testing-library. You can see there that now when I make, when I tried to get the text, the '(todoText)', then now we are going to have to resolve that in an asynchronous manner. Now, in addition to that, we want to verify that that has taken place within the mock because we are not sure at this point or we could be not sure. So, I'm going to go ahead and add an 'expect(mockCreateItem)'.toBeCalledTimes(1);'. That's the one time that we called it, when we made the addition of '('wash car')'.

And then also to be called, this is the data that was passed in. And we have a second 'expect' there to this, of the 'stringContaining('wash car')'. So now, let's go ahead and save this page here. And let's go ahead and open up our console. And let's go ahead and run 'npm test'. Let's see how we do. And as you can see there, we have a failure. So, let's go ahead and scroll up and see, take a look. And as we can see there, it's unable to find the text 'Learn spanish'. And that is because if we take a look, we only have 'wash car'. We use ' "Learn spanish";' simply to seat it. And so, I'm going to go ahead and change this here to '('wash car')'. I'm going to go ahead and save. And as you can see there now the test is passing.

So, I'm going to go ahead and close here my terminal. And I'm simply going to click that away. But as you can see, this can be quite important because anything, anytime you're building a real application, the likelihood of it having something that is asynchronous or having some major dependency on other block of functionality is a pretty high one. And you want to focus on making sure that your part is working as expected. And so, you can mock the other parts. So, this is one to dig into. Go ahead and replicate the work that we have done. Go ahead and add some additional modules, and make sure that you can call them and you can work with them even if they are not defined. That is, that you are mocking that functionality.

Reflecting On Your Skills In Front End Development

So, in this course, we covered a lot of ground. We started with ES6 and developing JSX web components. We started to look at the life cycle of those components and how to use React Hooks to control that lifecycle. We looked at keeping state between components using useState and useRef. And we looked at complete apps. We built bad bank, where we were looking at routes and routing and how that worked. And we built the shopping cart, which had a complete database in Strapi, and we used Postman as well.

And finally, we deployed our shopping cart into the Cloud. We built a static website using Create React App and npm build, which was one of those moments when everything is right. You know they've done it right, properly. It's a beautiful build that we've suddenly been able to build our app and deploy it into the Cloud and it's running. That's a great sense of fulfillment. I've really enjoyed this course. I hope you have. I'm sure you've got a lot in your repositories and in your portfolio, and you'll be able to show any potential employer that you know how to create web apps. Bye for now.