Testing React Applications with Jest

by Daniel Stern

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

React applications are the preferred way of delivering content to clients and customers over the web, but without good testing, knowing if your application works as expected is a matter of guessing and luck. In my brand-new course, Testing React Applications with Jest, you'll learn how to test React applications from the ground up. We'll learn how to install Jest and integrate it with a new or existing node application. We'll learn about running tests with Jest, but we'll also learn advanced techniques. You'll learn how to scaffold tests using mocks, and how to create automatic tests with snapshots. Finally, we'll learn how to test fully-featured React components using all the techniques I've just mentioned and more. By the end of this course, you'll have written and learned to manage multiple kinds of tests by coding along at home during the demonstration portions of this course. Full-course files are available for you to study and to use in your own projects. Before starting this course, you should be highly familiar with React applications. If you're not already familiar with React, I recommend my recent course Isomorphic React, which is available here at Pluralsight. If you're already a senior React developer, feel free to jump right in. Don't rely on guess work and random chance when building applications that are reliable, durable, and progressive. Master React testing with Jest today, and enter a world of testing opportunities.

Course Introduction

Introduction

Hello and welcome as we kick off this course, Testing React Applications with Jest. I'm Daniel Stern at @danieljackstern on Twitter, and in this clip, I'm going to introduce this course and tell you a little bit about it. So this course is a completely revised version of a different course by the same name that was released in 2016. Now, this new course consists of 100% new and updated content. Almost nothing has been reused. And everything about this course has been designed to fit in with the modern web development ecosystem that exists today. In that spirit, we've grabbed the latest versions of all the libraries we'll be using--React, Jest, and more. In addition, we've carefully noted your suggestions from the previous course, and this one has been built to exceed the expectations of any Jest learner. In the next clip, we'll be discussing the technical aspects of this course, including the demo project.

Why Take This Course?

So why take this course? Well, first and most obviously, knowledge of Jest makes you a more in-demand React developer. And once you've mastered React to a fair degree and worked on lots of projects with it, what more can you say about React? Being able to say, I also understand Jest, is terrific. And employers do notice this skill when they're reviewing your resume. Next, by taking this course, you'll be gaining skills related to the DevOps and QA fields. DevOps are responsible for facilitating the work of entire teams of developers, and QA is an in-demand position that ensures critical applications are running correctly. As a matter of fact, according to the 2018 Developer Survey on Stack Overflow, DevOps professionals are some of the best paid in the industry. So if you're hoping to join a DevOps team for developers working on React, knowing Jest and having taken this course is terrific. In addition, you'll be gaining experience with testing applications in general. So if in your current workplace, you happen to be working with an Angular app being tested by mocha, 90% of what you'll learn will apply, and these ideas even if you can't use them directly in an Angular app are very valuable just so that you can contribute to the discussion and let your coworkers know that you're very informed and understand the application's requirements well. Finally, and this is something more for individuals kind of more like me who may be more on the freelance side, but also to very senior developers and people running cool teams, in a phenomenon which I like to call Stern's Law after myself, all applications will eventually reach a size that no individual, even if it's that application's owner or sole creator,

can fully understand. That means according to Stern's Law, no matter what kind of application you're building, if you keep building it, and it keeps getting bigger, eventually you won't understand it anymore or at least not all of it. So by adding tests to your application as you go along, it's kind of like reinforcing the tunnels in a mine. It's less likely to collapse behind you and leave you with a big mess of code that doesn't work. This principle is good not just for, say, freelance developers who are working on maybe a larger project by themselves and need some additional support to help them understand it, or even if you're a senior developer, and you're looking at building a very, very complex application like maybe a big ecommerce site for a senior retailer, even though you have a team of experts, at the end of the day, it's going to get so complicated that you don't understand it. So by understanding how to do tests, you can avoid this phenomenon and reach that point of where you just simply can't make the application any more complicated much, much later. So I hope that sounded good. And if it doesn't sound like something that you're interested in, be sure to check out my other course, Isomorphic React, which will teach you all the React skills you need and after which, you may appreciate Jest testing just a little bit more.

A Look at the Demo Application

In this clip, we'll quickly discuss coding along with the demo application. So regarding the demo application itself, it's the exact same application that we built in the Pluralsight course, Isomorphic React. If you don't already have the course files from completing Isomorphic React, don't worry at all because the entire application can be cloned off of GitHub, which we'll be discussing in just a moment. However, to really understand the nitty-gritty of the application that we're building and what the tests are meant to accomplish, I do recommend that you complete the course, Isomorphic React. It's a great course and will give you a great leg up on this course. Of course, if you're already a React expert, then you can clone the application and proceed to the part of this course where we add the tests. So now to code along with me as we add the tests to this application, you're going to need the following technologies on your workstation. You'll need Node. js@9. 2. 0 or equivalent. You'll need your favorite text edition. Atom or WebStorm are both good examples that I've used, but you can choose any that you've successfully used in the past to make a JavaScript application. You need to have the kind of privileges on your workstation to install npm packages. So if, for example, you're working on a school workstation where you can't install any packages, you may have some trouble completing the demo application. If this is the case, you a laptop or see if you can get some increased privileges to install npm packages as necessary. Finally, you'll need to have the demo application running and ready to go. We won't be

building the application in this course, but we will be adding tests to it. So here I am at the demo application just so we can have a quick look and get refreshed on what we're working with. You can access it by going to the URL here at the top of my screen. If it takes one or two minutes to load, that's just because the app goes to sleep if no one's visited for a while, and it's not the app itself performing slowly. As you can see, it has a repeating list of components that represent a question, and these components have nested lists repeating the tags that they represent. It also has an additional view wherein the details of each question are displayed. So the app as we're looking at it was already completed in the course, Isomorphic React, but we will be adding a lot of tests behind the scenes. In the next clip, we'll get you set up with the course files.

Get the Course Files Here

Let's discuss where you can get the course files for this course. Now these files are available for download right here off this site or app, but the ones I'm about to show you on GitHub may be updated a bit more frequently. So if you already watched Isomorphic React and already have the application that runs on your computer, you're already done, and you have all the course files you need to code along. If not, you can clone the application, Isomorphic React, off this URL here. So here we are at the URL, and all you have to do is click the Clone or download button and clone it using GitHub. Or if you really don't want to do that, you can just download it as a zip. This repository also contains instructions on how to get the app started. So if you have a little bit of trouble getting it working, go ahead and read these instructions, and you should be good to go. Now though I believe strongly that the best way to learn is by doing and coding along is a big theme in all my courses, some users will prefer to have the completed files at their disposal including a test to begin with. And I can respect that. So at the URL shown, you can download the finished application including the test. So that will basically take you to the application state as we finish it in the last module of this course. And then, finally, like I mentioned before, probably the best course of action for those who have the time to do so is to just watch the course, Isomorphic React, and build the application. That way, you'll really be coming into this course fully prepared with all the background understood. Plus, it'll give you a good insight into React, which will be key as we'll be testing React applications in this course. And here we are at the course's website. And you can visit this URL if you want to watch it either before you watch this course or after. It's a great course, and it's also offered by Pluralsight, so if you have access to Testing React Applications with Jest, you probably have access to the course, Isomorphic React. In the next clip, we'll manage some expectations about what we will and won't be covering in this course.

Course Roadmap

Let's spend a few minutes discussing what this course will and won't contain. After all, I don't want you to spend your time watching this course only to discover that it did not contain a discussion that you thought might be present. So what will we be discussing in this course? So we will be spending some time getting to know the general architecture of testing--what kind of tests there are, why you would use testing, how testing works with teams, etc. You will also be given the opportunity to skip that module if you're only interested in the technical aspects, but I do strongly recommend that you watch it. Even if you already consider yourself quite skilled, as we all know, the key to mastery is to never forget about the fundamentals. We will be discussing Jest installation, running tests with Jest, and getting to know many aspects of the Jest API. So if you're coming into this course with little or no Jest knowledge, and you want to be leaving it with a really solid understanding of Jest that you can work in the workplace, you're in the right place. We'll be learning about mocking, which is a key feature of Jest and a few other libraries. We'll learn how it's done, why we do it, and then give it a shot ourselves and test the component while using mocks to help us. We'll be discussing and demonstrating three kinds of tests--unit tests, component tests, and snapshot tests, which are all in Jest's field of expertise. However, there was a lot to pack into this course and a very limited amount of time to do so. So here're some topics that we won't be able to cover. So we're not going to be talking about the basics of using Node or the basics of using JavaScript. If you're not familiar already with these, then I recommend you watch a less advanced course from Pluralsight's excellent selection of courses. We won't be covering how you actually build a React application. I'm assuming that you already know that. And if you don't, you can watch the course that we saw in the previous clip, Isomorphic React. So from this point on, I'm just going to assume that you understand the elements of a React application, as well as container components and an understanding of React Redux. Also, Jest has a large number of configuration options and command line interface options that are simply too large to cover in this course. If I covered them, the course would be double the length, and probably there wouldn't be a huge amount of value that you would get out just from watching them all one after another. So to find that specific flag you need for the command line interface, then you may still need to consult the docs. But this course will still be immensely valuable in helping you understand the context and content of the Jest ecosystem and its documentation. In addition, there are many tools that are related to Jest, for example, Enzyme and Mocha, which once again we just didn't have the time to discuss fully. This isn't going to limit your understanding of Jest in any way, but it's reasonable to expect that we might have spent a lot of time talking about Enzyme since they are often had in the same conversation, but our discussion

of Enzyme and Mocha will be limited to a few short clips. So what's the roadmap of things ahead? The module that comes right after this one explains testing in general. So we won't be talking about any Jest specifics or actually running a lot of code, but we're going to get familiar with unit testing, component testing, regression testing, and all the terms that you'll need to know. And if you're already very familiar with testing as in an expert, and you just want to know about Jest, like if you're coming to Jest from a different language already as a test expert, then you can skip from the end of this module to module 3 without missing anything. I was sure to include all the general information about testing in the next module so it can be skipped if you want. Now I recommend that you watch it, but the option is yours. After module 2, though, all the chapters deal with Jest and React. So we'll be learning about how to use Jest and then how to use it specifically to test React. We'll spend some time in a few different modules learning about the various kinds of tests that Jest excels at, including snapshot, unit, and component tests. At the end of the day, the Jest knowledge that you acquire will be used not only to test React applications, which it excels at, but for many general purposes at all, originally for me as is probably the case for you, I learned Jest to test React applications that I was building for a client. However, now, I reach for Jest when I have all sorts of problems relating to code to solve. There's a particular back-end API or a tricky function that just isn't working right, Jest is the first tool that I reach for. And, of course, it's the first tool I reach for when actually testing my React components as well. So hopefully you should be really excited about all the exciting possibilities that are to come in this course. So what can you expect in the next module, which is coming right up at the end of this clip? We'll learn about the advantages and disadvantages of testing. We'll discuss regression and how it can affect you in the workplace. We'll learn about the different types of varied tests that exist. We'll learn about unit tests, component and snapshot tests, and end-to-end tests, which Jest does not really handle, but it's important that you understand what they are so you understand exactly what it is that Jest cannot do. And lastly, we'll discuss some of the other testing tools that exist in the testing ecosystem that you will certainly come across at least once in your development career. It's going to be a fantastic module, so stick around, and let's learn about testing.

Understanding Testing

Why Testing?

Welcome to This module entitled Understanding Testing Concepts. Over the following series of clips, we'll be learning about all the ideas and techniques that are necessary to understand testing as a whole. These ideas do apply to Jest in that by knowing them better, you'll be able to use Jest better, but they also apply to a wide variety of other libraries and almost all test-related applications. As with any laborious undertaking, it's good to understand why we're doing something before we do it. Hence, why testing? What's the purpose? The first purpose of testing is that it prevents regression. So regression, which we'll be talking lots about, is basically a phenomenon where your website without you wanting it to goes back to a previous state that is worse than your current state. In other words, it stops working. Next, testing provides objective success criteria. By writing, for example, unit tests in advance and then making an application that fits into them, you can objectively say that the application has succeeded once all unit tests pass. Finally, anyone who's built a large application will know that a really big program with many different components and parts quickly grows beyond the scope of understanding for any single human being. That means to build a huge project that is beyond your own individual understanding or the understanding of a senior developer, you need more sophisticated tools to do so. Testing is one of those tools as it allows you to ensure the functionality of an almost unlimited number of components. So I hope I've piqued your interested with all these great reasons why you'd want to test your application. In the next clip, we'll get down into the nitty-gritty by asking, What exactly are tests?

What Is Testing?

In this clip, we'll answer the question, What are tests? Let's go beyond the cursory explanation that a test is something which tests your application and go deeper. Let's see what tests really are and what form they take. So a suite of tests is an application. That's a good way of looking at it. It's comprised of various bits of computer code much like your main web application. However, the purpose of this application isn't to deliver a user experience, but it only exists to check and verify the real application. Line by line, tests are fully of assertions about how your code will execute. We'll talk more about this idea of assertions, but they're basically statements like This will be true, or That will be equal to 5 or 6. When using Jest and making JavaScript applications, our test files take the form of JavaScript files, and they're committed with our repository when we commit our changes to source control. Finally, this suite of tests or this application is able to be quickly executed by your continuous integration software whether that's Travis or one of many other different valid tools. So at the end of the day, a successful test suite will be verified automatically every time the application is updated in production or any time the code base

changes really. Rather than a human looking at all the code and checking every subpage of the website manually, the tests are instead run instantly and automatically. So let's flip the question around. What if tests didn't exist? This is a question that often has me suddenly awakening in the middle of the night in a cold sweat. It's such a terrifying question because all of the following things could happen. So without tests, every time you changed your application, if it was a mission critical production application, someone would have to check every single page every single time. There's no way without tests for you to know if your web page has stopped working except for someone like a QA professional to check. Next, when you commit new code to this repository, you had better check every single page and component as well because there's no way without checking to know if your code has broken someone else's part of the application. There's no way to measure how correct a piece of code is without tests. Someone simply has to look at it and say, I guess that works, or It seems to work and move on. With unit tests, you can say that the test absolutely does come up with the correct value for any of these given scenarios. Now this sounds bad, but it actually just continues to get worse. Once your application becomes large, once it becomes a major-sized application, manually checking for these cases of regression becomes too burdensome to even do. You end up in a situation where paying your QA team to check every page can cost hundreds or thousands of dollars for every update. Finally, adding new features becomes just too expensive, and the application stops growing. It simply ceases to evolve and continues to live in a half-life state until it is no longer profitable. Now ladies and gentlemen, do not worry because this nightmare scenario is just a fantasy. Tests do exist, and they are there to help us prevent all these things. However, they also have disadvantages. So in the next clip, we're going to look at the advantages and disadvantages of tests.

Advantages and Disadvantages of Testing

So let's state the obvious here. Here I am presenting this course on testing. Obviously, I wouldn't be here if I didn't already think testing was really great, and you should be doing it. However, I would not lie to you and say that there are no disadvantages, that it's all simply easy wins in the world of testing. Let's discuss what the advantages and disadvantages are. So what are the advantages? As we've mentioned, it prevents regression. It reduces the need for humans to manually verify the application. It allows you to verify corner cases like very unusual inputs that you wouldn't see very often in production but still need to be tested. A side effect of this is that it allows the everyday developer, in other words, you, to focus on the current tasks that they're doing now. It removes the burden of worrying about past components and regressions that could possibly come up. And, finally, by using tests and unit tests and building an application up in this

way, we could make applications that are just plain more complex than applications we could build otherwise. But what's the other side of this coin? Why would we not want to have tests? Well, first off and most obviously, it's more code to write. Tests are code, and you're writing them. And like normal code, tests also need to be debugged and maintained occasionally. Sometimes components will change, and the tests need to change too. Next, like we're seeing now with tools like Jest but also Jasmine, Mocha, Enzyme, Protractor, the list goes on, testing is just more tools that developers like your or your team of developers if you're the leader need to be able to use. Additionally, you're adding several additional dependencies to your application. Usually it's not a huge amount for a JavaScript application, but it still is more dependencies, and that's not really something anyone wants. Plus, the tests that work on your computer may not work on your cloud host's environment when running your continuous integration tests. Honestly, you don't know until you've tried it. And when you just want your application up and running, this can be an unnecessary and tiresome roadblock. Finally, and any senior developers will agree with this here, is that you actually have to use tests and use them right for them to be of value. So often, you'll be in a workplace and see that when someone makes a change that causes a regression, rather than fixing it, they'll just disable the test. Disabling the test is such an easy and convenient way to make the problem go away. The only downside of this is that it doesn't actually do anything to solve the problem. It's sometimes hard to tell whether a developer has actually used the test for some real purpose or simply created to make it look like the application was actually being protected. At the end of the day, as a senior developer, you have to work with junior developers and help them learn how to use tests properly. The rewards are great, but the time investment is real. Finally, as the old axiom goes, humans are easily tired but computers are easily confused. You will often have situations where a tiny change or regression that really has no literal effect on the user will cause your test suite to break. Your test suite does not care that the only regression is incredibly small and of no real significance. As far as it's concerned, nothing works. The continuous integration will not pass, and the application is just broken. Thus, maintaining tests requires frequently fixing these small error notices. Now as we'll see in my opinion at least the advantages of testing greatly outweigh the disadvantages. In the next clip, we'll learn all about regression and how testing helps us prevent this pesky hobgoblin.

Understanding Regression

As promised, we'll now explain how regression works. Regression is a fascinating phenomenon that can impede and has impeded applications of all sizes. But let's imagine the following scenario. Developer A, in other words, you, add a new feature to an application. This feature

improves the views known as View X and View Y, so now they work better. As far as the application's schematics are concerned, View X and View Y are correct. Now developer B, in other words, that annoying person over there, modifies this feature which you added. You know, probably they were told to do so, that's just their job, but they've changed the feature. And unfortunately, this other person, they know about View X, which is, say, the main view, but they're not even aware---they don't even know View Y exists. Maybe View Y is some obscure payment or credit card processing page. We can't really even expect a developer to be aware of every single page on an application. This is hardly anyone's fault. So the developer looks at the big view, View X. It looks good. They commit the changes and take a long lunch, give themselves a nice big pat on the back. Unfortunately, this has resulted in View Y regressing to a state that is not intended. Very likely, it's in a broken state, and without tests, no one is likely to discover this until someone actually goes to that page and reproduces the bug. Regression is just fiendish, ladies and gentlemen. It actually keeps me up at night. As William Shakespeare once penned, "The evil that men do lives after them. " This is regression, folks. However, ladies and gentlemen, there is a happy ending to this tale because testing, which we're learning about, actually prevents regression from happening. Here's how. So here's a similar scenario. Developer A adds a new feature, and Views X and Y are improved. They also add relevant tests, which just confirm that View X and View Y look the way that the developer last saw them. Once again, X and Y are correct. And all tests are passing. Now Developer B comes along, and they once again modify the feature. They know about X, but they're not aware that View Y even exists. However, when they make this change, they notice suddenly the test suite starts failing, and tests relating to the view known as View Y are indicating that View Y no longer looks correct. Oh my goodness, thinks developer B. I didn't even know that View Y existed, but now I do. Obviously, I can't commit this without fixing View Y. So first I'll fix that, then I'll commit the changes. Folks, regression has been nipped right in the bud. As I hope I've demonstrated, regression is not just an annoying problem. It's a costly problem for large companies. So just by learning to prevent regression, you can create so much efficiency in your organization, it's almost a good enough reason on its own to learn testing. Of course, the other advantages are also many fold. In the next clip, we'll be exploring and demystifying the different kinds of tests.

Types of Tests

There is more than just one kind of test, and interestingly, some different kinds of tests do completely different things and work completely differently than other kinds. In this clip, we're going to become experts on these different kinds of tests. Now the first kind of test, the one most

people will be familiar with, is a unit test. A unit test tests a single function or service. So, for example, a function that takes a number and returns all the factors of that number or if the number is prime or not is a perfect example. To do a unit test like this, you need some kind of test runner, a simple one like Mocha or a more full-featured one like Jest. A component test verifies the functionality of an individual part of an application. Think like a widget or a drop-down list. Components can have lots of different functionality. So taken together, a suite of tests testing one component is supposed to cover all these different functions from user interaction to how it looks. Your component is just covered. Doing this requires more sophisticated tools than Mocha or a simple test runner often offers. So we need Jest and sometimes more sophisticated tools like Enzyme, which we'll be discussing. So a snapshot test is a special kind of test that we have a whole module devoted to later on. It basically tests a single component like a component test, but it can't really verify the functionality of that component, or you can't really use them in advance and then build your component to that. It just protects a component from regressing. But since regressing is so expensive and annoying, this is almost worthwhile in itself. Now to do this, this is basically one of the specialty features of Jest, which is why we have a whole module coming up on it. Lastly, a special kind of test exists called an end-to-end test. An end-to-end test really tests something from the perspective of your user. And while we won't be covering them very much in this course since they're not something that Jest does, it's so important that you understand what they are, and you can at least explain why Jest doesn't do them and how they're not related to Jest. Let's take a bit more time to get familiar with these different types of tests starting with unit tests. So unit tests just verify the functionality of a particular class or a given method. They're simple to understand, simple to write, simple to execute. Unit tests are easy to understand because what they do is so logical. You have a function that adds two numbers, and you write a test to say, If I call this function with 2 and 2, is the response 4? It is, great, so the test passes. These are all very logical when you think about them. Unit tests are unique as they're some of the only tests that can actually measure the correctness of the application logic. You might have very complex logic that can output, for example, numbers that are too difficult just by looking at them to see if things are going exactly right. So unit tests can let you say, Here's this component, in the case of all this unusual different kinds of input, it always returns the exact right value. And, finally, a unique aspect of unit tests is you can write them before the components or the functions in question. So you can have a unit test that tests a function, and we don't know anything about this function except that when 2 is the first argument and 2 is the second argument, the value that's returned is 4. So we can write this test and then write the function in a system called test-driven development. TDD is awesome, and unit tests are the kinds of tests that are most often used to achieve this. Next up, we have component tests. Component tests may seem similar to unit tests

in that a component of your application is like an individual unit of it. However, when we say component, we mean something that has more than just input/output functionality. It may be able to render HTML or receive user input via DOM events, and so we look at them as a distinctive type of test. So component tests verify that a particular component looks right and it works right. In other words, the expected events are broadcast when users interact with them in a particular way. Component tests are very sensitive to small changes. So if you have a component that has five levels down another nested component, and a capital letter changes to a lowercase letter in that component, you're likely to see some component test fail. In this sense, they're like a canary in the coalmine of your application being the first point of warning when something is even a little bit off. Component tests provide a great defense against regression because, like we just said, they're so sensitive. If every component in your application has a large number of component tests making sure that functionality does not change, then it's actually really hard to regress without ignoring your tests. In addition, component tests can measure more complicated things. For example, your component looks one way, then a user clicks a button on that component, then your component looks a different way. Finally, it's important to note that traditionally components don't verify the interaction between two components. This is the realm of an end-to-end test. Now let's talk about snapshot tests. So a snapshot test is it's like a subtype of a component test. It is once again verifying that a component does not regress. However, and most developers are pretty happy when they hear about this, snapshot tests are generated automatically. You don't even have to write them. All they do is they verify the last time a snapshot was run, is the result the same as this time that a snapshot is run? So they're so easy to put together because it's all automatic, and they actually do a lot to prevent regression. In other words, they're a really cool tool. Of course, their downside is that if even the tiniest little thing changes, something that no one ever intended to be of any importance, the test will fail, and you have to go back and fix it. Now let's briefly demystify three other types of tests, tests which Jest is not really designed to do. The first is a performance test. So a performance test measures how long it takes for a block of code to execute. They're a good tool if you have an application that you wish were running faster, but you don't know why it's running slowly. A performance test can help identify the cause of the bottleneck. And additionally, while your application may perform quickly on your computer, it may perform slowly up on the cloud. So performance tests allow you to find out how your application performs in various different environments. Next are coverage tests. Coverage tests are like a test for your tests. A coverage test looks at what code is visited in your application when your tests are running and tells you how much of your application is actually covered by your tests. If a whole lot of your application is not covered by tests, then your tests have effectively failed. It's like there's so much that your tests aren't even touching, how do you know

that they're protecting your application? As Aristotle said, "A Jest which will not bear serious examination is false wit. " However, we can seriously examine this Jest in this course. Finally, and although Jest can't do these kinds of tests, I felt it was so important just to talk about them, here are end-to-end tests. So end-to-end tests measure the functionality of an entire application. They're often run in a virtual or headless browser that really simulates a user clicking in your application looking at it and experiencing it from beginning to end. End-to-end tests work by creating a scenario, for example, a user logs in, checks their account balance, and logs off, and then running these in some kind of virtual browser. End-to-end tests are rather different than writing unit tests. In fact, you might say there's a fine art to it and that it is quite challenging. However, at the end of the day, end-to-end tests provide a great assurance that the application really works. The key paths to your revenue are really there and really functioning. In the next clip, we'll summarize what we've learned in this module.

Summary

Let's wrap up this chapter on testing theory. So tests can prevent regression. They can make sure the app functions correctly. And they can measure the performance of an application on different platforms. The bigger the application and the bigger the team, the more valuable tests are. Just one person making one small thing, tests aren't so great. A thousand people making a 10, 000-page application, tests are great. We learned that regression is a costly and tiresome phenomenon. But we also learned that you can prevent regression by using tests. And when you do so, all features will break less often. We learned about all the different kinds of tests. We've learned about unit tests, which are very small in scope and which Jest excels at. And we even learned about end-to-end tests, which are a complex type of test which Jest does not do. So what's coming up in the next module? In the next module, we're really going to dive in to Jest now that we know where it fits in with all this testing theory. We'll learn what it is and how you can use it. We'll ponder the question, How does Jest differ from other frameworks? We'll take a look at Enzyme. Is it right for you? And much like testing has its own advantages and disadvantages, Jest has its own ups and downs just even within that own library. So we'll discuss those so you know what you're getting into next time you choose it for your project. It's going to be great. I'll see you there!

Introduction to Jest

Introduction

Hello and welcome to this module entitled Introduction to Jest. In this module, we'll learn all the information about Jest that you need to know to be an effective Jest developer. As William Shakespeare once said, "In Jest, there is truth. " Will you find the truth of whether or not your application is functioning correctly by using Jest? I certainly hope so, but let's find out. So, what is Jest? I'm sure you already have a pretty good idea of what it is, but what is it exactly? So Jest is a JavaScript library installed via npm or Yarn and run via the command line. Jest fits into a broader category of utilities known as test runners. It's similar to other populate test runners like Jasmine, but it has a lot of extra useful features. And Jest is a tool that's made by a team of excellent developers that include many members of the React team. So in your mind, you may correctly already have Jest and React associated. And this is the main reason why. It's that they're both built by teams that largely overlap. But, of course, in the world of web development, there're many tools, and we need to understand where Jest fits in among all of them. In the next clip, we'll do just that.

The Jest Testing Ecosystem

So to understand how you're supposed to use Jest to effectively test React applications, you have to understand where Jest fits in among all the other tools that you also have to use to achieve this goal. So what are the main testing tools that we have to be aware of when we're testing React applications. So the first one is Jest. Nothing fancy there. It's the library we're talking about throughout this whole course. Enzyme is another very popular library that is used for testing React applications. We'll be talking about it more later, but suffice it to say it works with Jest and has many advantages, as well as disadvantages. Finally, Jasmine and Mocha are names that you'll hear often during discussions of Jest. These are tools that Jest is built on top of. So what are Jasmine and Mocha. So these are both very similar test runners that both organize tests into groups called describe and it blocks. Or sometimes you might hear them using their alternate names suite and test. If these already sound familiar, that's because Jest also uses describe and it to wrap its test blocks. So the way that these tools work is that whenever you run them, they run all your tests and look at all the assertions inside those tests. So by invoking Jasmine with the command line, you get a quick readout telling you if all your tests pass or not. However, these libraries do not include mocking or snapshots, features that we'll be talking about quite a bit in this course. Now let's talk about Jest. So Jest is built on top of Jasmine and Mocha. So almost all the features that they have, Jest also has, but it has a lot more features as well. Jest takes this great framework built by Jasmine and Mocha and adds snapshot testing, mocking, and a lot of

other really cool features. In addition, Jest has an assertion library that one could argue is superior to that available in Jasmine and Mocha, as well as a command line interface that many would say has a lot more features that are advanced and useful. Lastly, and this is the point you're going to hear me make a few times, Jest works great for React applications, but it also works great without React applications. It works really well in both situations, so calling it a library to test React apps isn't very accurate. It's more accurate to say that it's a great testing library built by many of the team members of React that happens to work really well for testing React applications. Lastly, what is Enzyme? Is Enzyme a tool that we use to replace Jest or use it within Jest? What's going on here? So Enzyme's not a test runner like Jest. It doesn't provide a simple output indicating if all your tests pass or not. But it has tools that Jest can use to test React applications. So the ultimate purpose of Enzyme is to take your React component and kind of spit out a DOM tree or some HTML that you can run assertions against. This works a lot like a utility we'll be using a bit more called React Test Renderer. They basically both do the same thing, but Enzyme has a lot of fancy bells and whistles. So Enzyme is potentially very powerful and can be very useful, but it also has quite a few drawbacks. So here we are at the GitHub repo for Enzyme. As you can see, Enzyme is a fairly popular library with over 13, 000 stars. Unfortunately, it has a lot of open issues. In other words, it has quite a few bugs, and when you have a library that you're trying to use to assert if your code is correct, a bug within that library can be a real problem because now you can't even tell if it's a problem with your application or with your library or a different component. So in my opinion, when it comes to testing, less is more. So I try to steer clear of Enzyme because in my personal experience, it has issues that actually make your apps harder to test. So I hope they improve it in the future, but for this course, we won't be using Enzyme for the reasons I just explained. In the next clip, we'll be comparing Jest with Mocha a little more closely.

Jest vs. Mocha

So let's look more closely at the ways that Jest improves over Mocha. You may find this to be a little bit academic since we've all already agreed that we're going to be using Jest for this course. But rest assured, when your paycheck depends on you explaining why you chose to use Jest for a particular application and not Mocha, you'll be glad we spent 2 minutes having this chat. So Jest versus Mocha. First of all, they both run tests. They're both test runners, so their ultimate goal is using a command line, you invoke them, and they return a simple value, a 0 or a 1, if all your tests pass or didn't. In this way, they're the same, and you can kind of swap them out as your test runner pretty easily in applications. Both Jest and Mocha can run asynchronous tests, which we'll

be talking about later in this course. But neither of them have any problems actually running assertions against something like a service that might take longer than one computation cycle to complete. So when you use Jest, you automatically get spies, which in Jest are often referred to as mock functions. Mocha doesn't have this feature, and you have to add it using another library like, for example, Sinon. It's a bit more work, and things are a bit less consistent because in every Mocha project, you may have a different spy library. Additionally, Jest supports snapshot testing by default, which we have a whole module on, and I'm fairly confident you probably came to this course mostly to learn about it because it's kind of the cool, exciting feature about Jest. And Jest can do it, but Mocha cannot without some additional library. Lastly, the mocking of modules. This is a bit feature of Jest, and once again we have a whole module on mocking, but Mocha doesn't do this feature. This is well beyond what Mocha is intended to do. Mocha is really just the test runner, but as you can see, Jest does a little bit more relating to your files. It mocks your files. It comes with its own spy library. And all in all, it's more full featured. In the next clip, we'll talk about Jest versions.

Jest Versions

So Jest includes two basic tools--Jest itself and the Jest CLI or command line interface. Even pro developers can confuse these, so let's get it perfectly clear what they are and what they do. So Jest is the actual test runner. It's the library. You call Jest, and it tells you whether your tests are passing or failing. Additionally, it provides watching features and other things. Jest is the library. Jest CLI is a tool that you use to run Jest. So Jest CLI doesn't actually run any tests. It just configures Jest based on whatever arguments you gave it, and then Jest runs. This is important because Jest CLI is the tool that you and also the continuous integration software you have will use to actually get Jest going. So this is important because these are actually different libraries and in many cases are installed differently. Having one version of Jest and a different version of Jest CLI can lead to truly puzzling errors that can take a very long time to debug. As you can see, versions in Jest are actually quite important. The latest version of Jest at the time that I made this course was 22. 4. 2. This is important because Jest is a pretty dynamic library relatively speaking. And each version has slightly different API signatures or methods. I don't want you to think that everything you saw in this course is exactly how it's going to be when you go out to the workplace because that's simply not the case. Versions in the 0. anything family and 1. anything family work fairly differently than the version that we're doing in this course, which is the 2. anything family. Now you may be thinking, Why would I use these previous versions when I have the new and better version at my disposal? Well, I wouldn't be surprised at all if when you were to

go into a regular workplace situation that was using Jest if they were using one of these older versions because the setup they have could be in place for years. So generally what you'll learn is useful, especially the overall concepts about testing, mocks, spies. But there's going to be a tiny variation in the APIs or sometimes the automatic mocking behavior, little differences, so be ready to deal with those. So if you ever have the opportunity to choose the actual library or version that will be used for a project, do everyone a favor and choose one of the latest versions. As a wise developer once said, "No one ever got fired for using Jest. " In the next clip, we'll look at Jest and React and observe the connection between these libraries.

Jest and React - What's The Connection?

So in what way are Jest and React connected? Many people may think that Jest is a library simply for testing React applications. This is not accurate, but as you can see, it's founded with a grain of truth. So when it comes to choosing your test runner for your React application, Jest makes the most sense because it's recommended by the React team in the React documentation. So in many cases, the connection is so obvious because the decision has already been made for you. If your architecting a React application at a major corporation, and the documentation tells you right there that you should use Jest, you really have no reason not to. It's pretty risky to choose some other software that might not work, whereas if you choose Jest and things don't end up going great, you can say, This was a technology that was recommended by the team that developed the software. I don't see how I could have made a more accurate choice. However, Jest has an extensive feature set, and it's used to test many other applications. From personal experience, whenever I'm running anything in JavaScript, whether it's an API or some kind of artificial intelligence network, if I have a problem with it I can't figure out, Jest is the tool I reach for to fix that problem. So don't think that just because you always hear Jest and React together that you need to be using a React application to have Jest. Now it is true that both of these libraries are connected to Facebook. Facebook employs many of the developers, the core developers of React and, therefore, of Jest. So in practice, you're going to find them together really frequently. In any job or workplace situation when you're working with Jest, you'll probably see React involved. Finally though it used to be the case that these libraries were under more restrictive patent grant licenses that really were not very good and were not very popular among the community, they're now both licensed under the MIT license, which doesn't quite mean you can do anything, but it's a generally accepted standard, and in the development community, it's an agreed-upon good license for libraries like these. So in terms of licensing, they both license the

same, and the license should meet your needs as the developer. But that's a lot of academic stuff. In the next clip, let's discuss how Jest actually works in the field.

Practical Jest Usage

So we've talked about the theory behind it, but what about the nitty-gritty? What does Jest actually look like when it's implemented in the workplace? So, here's an outlay of how Jest is actually used by developers, companies, and by extension you. So a script that kicks off the developer's development cycle, an npm script, is run. And this script triggers the Jest watcher. The watcher, which we'll talk more about later, automatically watches for changes to the file and notifies the developer in real-time if they've broken any components. And this script that sets the watcher up would probably be set up by a senior dev or DevOps. On the flip side, another script also probably written by DevOps is run by the continuous integration or CI software that you have. This script runs Jest, but unlike a developer who is just looking at the test results and adjusting their code appropriately, if an application fails its test suite, your CI software can be configured to reject the update or roll back to a previous working update. So while your devs use Jest in a somewhat subjective sense, your continuous integration software uses it in a very objective sense. Meanwhile, senior developers are constantly reviewing the tests and test coverage reports to make sure that the tests are actually running and that they're actually effective. It's easy as we'll see to write many tests that don't actually do anything, and unfortunately there's no substitute for the care and attention of a senior developer in preventing this. At the end of the day, the end user, your customer, is never even aware that you used Jest. The only thing they know is your application works like a charm. So as we've learned, there is kind of two different skillsets that you can have for Jest. As a junior developer, you need the following skills, all of which we'll be covering. You need to understand describe and it blocks, also known as suite and test blocks, and how you can use them to structure your tests so they're clearly readable by others. The junior developer has to understand eventually which tests prevent regression or the application functioning wrong, and which tests are just window dressing, in other words, tests that don't do anything except look good. Junior developers need to have practical strategies for fixing regression when a change they make causes a different component to break. And if you're a little confused about this term, regression, don't worry because we have a thorough discussion about it coming up. And, finally, a junior developer just has to write some good robust tests that make sensible use of the assertions and the APIs that Jest provides to make our tests useful. In other words, writing a test that works but requires all kinds of hacks and workarounds and is very difficult to read is not an entirely successful endeavor. On the other

hand, a senior developer has to know all kinds of different Jest skills. We'll be discussing all of these, but some of them are so low level that to really understand them, you have to visit the documentation after you've watched this course. So a senior developer has to configure Jest via the command line. So they have to understand all the features of Jest and the ones that make sense for their application, and there are a lot, and then understand also the configuration necessary to set that up so it's the Jest configuration that all the junior devs have on their machines. A senior dev also has to take these and package them into npm scripts that are used not just by developers but by continuous integration as well to make sure that your app is working. As we discussed previously, a senior developer has to be able to look at tests and code coverage and advise their superiors or the developers working under them if the tests are actually effective or not. Finally, as a junior developer moves on to being a senior developer, one of the most important skills is writing tests that don't merely prevent against a regressed state or create a quick snapshot, but tests that really express the author's intention. So if you're just starting out, you're going to want to focus more on the skills on the left. If you're a bit better, you'll still have to master the skills on the left, but you also might want to start thinking about the skills on the right since you'll need these as well. In the next clip, we'll talk about common pitfalls that can occur when using Jest.

Common Jest Pitfalls

As the old saying goes, if you ask the guy who's selling encyclopedias if you need an encyclopedia, he'll say, Of course you do. And if you ask the guy who's teaching the Jest course if you need Jest, I'll say, Of course you do. Jest is great. But in fact, there are disadvantages and pitfalls that go along with using Jest. So the first and most obvious pitfall is that tests are simply not written. This may seem like a humorous situation, but in fact it's very easy for devs doing day-to-day work to simply not write tests because they're eager to move on to making the next component. I mean, after all, it looks right. Everything seems to work right. Why write tests? So senior developers need to work closely with junior developers to teach them the importance of writing tests and get them into those good habits. In addition, a big problem is that if your tests aren't integrated with version control, then devs who are a bit lazy or not methodical with their tests can check files into the repository that are actually broken. And then when new devs run the application, they see that tests are already broken, and they see, Well, I don't have to worry about my tests passing since the suite is obviously already broken. No one seems to really care if it works or not. This is actually a real problem that I've seen happen time and again. Another problem is Jest isn't integrated into devs' workflow. So this is the job of a senior developer or

DevOps to make sure that, for them, they shouldn't even need to start Jest or anything. Just whatever script they use to get started should also start Jest. If it's hard to run Jest, they you can hardly blame your devs for occasionally forgetting it or skipping it so they can move on to the next exciting component. In the same vein, sometimes a dev will update a service only to see that it has broken tests written by a previous developer in the past. Rather than figuring out what the problem is and fixing those tests, it's far easier to simply mark the tests as skip or put some kind of superficial fix on it kind of like a bandage on the solution. It doesn't really fix anything. And when devs have a lot of pressure to deliver completed components and services at the end of the day, you can hardly blame them for doing so. So once again, senior developers need to be vigilant and teach developers the skills to actually fix problems rather than just pave over them. Additionally and similar to not being integrated with version control, if your tests aren't integrated with your continuous integration or your production server, then you're really missing out. Probably the most important feature is that when all human eyes fail, when the senior dev is on vacation, and someone tries to update the application in a way that would break the previous functionality, the continuous integration software has to be there to say, No, this request has been rejected. These tests have to pass. And at the end of the day, this is a good thing since it kind of takes the responsibility of making sure that the application works a little bit off the shoulders of the senior devs and puts it in the eyes of the software, this objective software. If you have a million carefully maintained tests, and despite all of this, the application still breaks in some small way when a change is pushed, no one can say that you were not at least cautious and observing best practices when this happened because if you have lots of tests and are integrated with continuous integration, this is basically the best defense that you have against failure, though it is not perfect. Finally, the last big problem is that tests that are written don't actually protect against the errors that are important. You might have lots of tests that make sure that your CSS styles look pretty good, but no one's actually ever written any tests to make sure that entering just any information into your credit card API won't result in a product successfully being sent out. So throughout the remaining modules, which are much more practical, and we'll really be writing some code in those modules, it should be really exciting, we're going to learn about how to fix all these Jest pitfalls. But when you're writing code, remember, if you fall into even just one, the usefulness of Jest is going to be greatly diminished. In the next clip, we'll wrap up this intriguing module.

Summary

So what did we learn in this module? Well, we learned that Jest is a modern library built on top of other libraries that are already very good, and it is developed by the very same team that developed React. So not only are they a great team, but they're especially great for making a test runner that will work well with React. The number one cause of Jest headaches is incorrect versions since Jest tends to vary in many minute yet significant ways from version to version. So being aware of what Jest version you're working with is one of the best skills to prevent surprises and unexpected bugs. We learned that ultimately Jest is just like Mocha but has a ton of extra features. So if you already understand Mocha or you're trying to explain Jest to someone who already understands Mocha, just explaining this really makes things a lot clearer. Lastly we learned that Jest really comprises two sets of skills, the day-to-day writing test skills for junior developers and the high-level test verification and maintenance and configuration skills that a senior dev needs. Coming up in the next module, we'll get our application up and running by installing Jest and running it using the command line. We'll watch for changes in our application so our tests run whenever we change a file. We'll write our first simple test, then build variations upon this theme throughout the rest of the course until we're writing pretty advanced full-featured tests. We'll learn about the zany, the wacky BeforeEach family, and the other globals that make up the supporting cast of your Jest application. We'll cover skipping and isolating tests, a basic feature but one that every Jest developer should know. And, finally, we'll cover async testing. And you may think you already know quite a bit about async testing, but when we look at the brand-new ways of doing it that have just become available in the past year or two, you might really be impressed. So it's going to be a great module, and I look forward to seeing you in it shortly.

Test Running with Jest

Jest Installation

In this module entitled Fundamentals of Jest Testing, we'll be applying the basics of Jest from getting it installed to writing a few tests. As everyone has said at one point or another, "It will probably work. " And if you've ever said this, you know that these words inevitably come before something does not work. So how do we make sure it'll work for sure? We're going to begin this chapter by, of course, installing Jest. Jest is installed via npm like a lot of other libraries. You can also install it with Yarn and some other tools, though we won't be doing that here. Now the

version of Jest that you install locally as in in the Node module's folder in your project should determine what version of Jest is used. But in practice when you use the CLI, you don't know whether it will call the local installation or a global installation if you have one. It's just not consistent across every machine. And as we'll see in the next clip, after we install it, the installation will make changes to package. json, which our continuous integration software actually uses to run Jest itself. So in the next clip, we'll be getting Jest installed. Make sure you have your text editor ready, and let's jump right into the first code-along demo of this course.

Jest Installation - Demo

Alright, are you ready to get in there and write some code? I sure am. Here's what we'll be doing in this short and relatively simple demonstration. We'll be adding a local Jest installation to the isomorphic React application. We'll be doing this with npm in a process that I'll demonstrate and you can replicate. We'll also install Jest CLI on a global level. And so we'll sort of get a feel of how the CLI is different from Jest itself. And then, finally, we should be able to confirm that our local Jest installation and the CLI we installed globally are interacting. So here I am inside WebStorm. As you can see, I have the isomorphic React application open, and this is the same application you can get by going to the link that was in the introductory module. I have also switched to a branch called jest-demo. You might want to switch to a different branch and also call it jest-demo so it doesn't have a conflict with the branch jest, which is actually the official finished version of this code that we're writing. So let's begin by installing Jest CLI. So we'll type npm install -g jest-cli. You'll notice I didn't include a version number there. This will just give you the latest version of Jest CLI. This is most likely to be compatible with everything. If I advised you to install an older version of Jest CLI globally to be more compatible with this project, it could reduce compatibility in your other projects, so that doesn't make a lot of sense. With Jest CLI installed globally, we can now type jest into the command line. So if you did that all correctly, you'll get this message here, No tests found. So that's good because we haven't written any tests. Now in the case of this application, we don't have any local Jest installation. When we run Jest CLI, it may try to use a global Jest installation to run our tests. However, if there's a local Jest installation, it will just use that, so let's install Jest locally and also save it. We'll say npm install --save. Now you could argue that this could be a save dev since this is a development type module, but since this module is also used by the continuous integration server, I just put it in the normal dependencies, but that's almost a personal preference. So we'll say npm install --save jest@22. 3. 0. And in case you're wondering, that's just the version that is the most latest when I'm making this course. There isn't anything special about that version of Jest that makes it better than the others. But if you're

coding along here, you'll want to use that version since newer versions may have different APIs, and what I'm saying might not make sense relative to newer versions of Jest. So now we have our local and our global install. We can type jest again, and we don't get any error. So that's good. In practice, you'll probably not spend very much time installing Jest. Odds are, this will already have been done by DevOps. But if you're DevOps or you're just a freelance developer making your own projects, that is how it is done. Join us for the next clip where we'll continue to add testing functionality to our application.

Running Tests

So in this clip, we'll be learning about and demoing running tests. There aren't a whole lot of tests to run right now because we haven't written any, but we'll learn to run the tests first, and then shortly, we'll write some really cool tests. As Benjamin Franklin said, "Well done is better than well said. " And saying that the application works is one thing, but making tests that do actions which verify this statement is all the better. So what's going on when we talk about running tests? So tests are run by using the Jest CLI as we saw. When you type jest in the command line as we demonstrated, this actually runs any tests that are in your directory. As we'll find out, there're various different kinds of configurations you might want for your tests. Certain tests may want to be run only in your local development environment, and in some cases, you may want the tests to be watched and run every time the file changes, which we'll also be demonstrating. So generally these various configurations are stored in package. json as an npm script with names such as test or test. watch. And on a practical level, it's important to know that the only component of your workflow cycle that's really caring about running the test is your continuous integration software. Really, your developers are more concerned about watching the tests since they're making these ongoing constant changes, so every time they make a change, they have to run the tests again. That doesn't sound very efficient. So when we're developing, we watch tests, and when we push to our centralized repository or production server, that's when we run the tests. So in this brief demo, we'll be doing the following. We'll be using Jest to run tests through the command line. As you saw, we already got a good idea of what that's like in the previous clip so we don't have to spend too much time doing that here. We will be updating our package. json to help us run Jest by setting up some shortcuts that help us run it. And we'll also take note what happens to the terminal after the tests are run. So here I am back in the text editor. As we saw, typing Jest runs the tests, of which there are none yet. So we want to immortalize this in an npm script. So I'll go to package. json, make a little more room there. So here in my scripts, you can see I already have a script for getting it started and getting it going on production. Let's add a new script called

test. And this is all we need for now, so we'll just say jest and save that. Now when we type npm run test, we get an interesting result. We get all these errors that appear on our command line. Why? Well, as you can see, when we ran Jest, Jest didn't see any tests. This is equivalent to an error, so Jest actually exited with an error code. That's why we got these errors because our terminal when the thread finished said, Whoa, this didn't end with the expected success code. This is the same method that is used by the continuous integration software to determine if your software works or not. If Jest ends with an error code, which it'll do if it does not find any tests, cannot run properly, or finds tests, runs them, and some do not pass, it will exit in such a way that your continuous integration blocks the update. Very, very cool. But you're probably thinking to yourself a whole lot of good being able to run Jest is without writing any tests. Well, I agree, so in the next clip, we're going to begin writing actual tests for our application. It's going to be great. See you there.

Creating Test Files

In this clip, we'll discuss creating test files. We'll actually be creating them in an upcoming clip later in this module. So test files, how are they identified? How does Jest know what's a test file and what isn't? So the first rule is that any files that are found within any directory with the name __tests__ is considered a test. So if you put a JavaScript file in one of these folders, Jest will try to run it when you call Jest for better or for worse. The second way is if you name any file anything. spec. js or anything. test. js, Jest will also recognize them. So that's right. When you run Jest, it searches the name of every folder and also the name of every file in your whole repository. So as you can see, you can have all your tests together in a tests folder or just anywhere in your application, if you give it a. spec or. test extension or even a combination of either of these. My personal preference is to name my test files. spec. js and put them next to the files they're testing. But is this the only way or even what everyone agrees is the best way? Definitely not. The argument on whether tests should be in their own folder or in the same folder as the component they're testing is by no means concluded. So rather than telling you that you should or shouldn't do this, let's just look at the arguments both ways. In the red corner, we have tests in their own folder. What are the pros and cons of this? So it's pretty easy to distinguish between test and non-test files when all the test files are in a folder called tests. On the other hand, you can end up with relatively unrelated files sharing a folder. For example, you might have a test for your loginService and a test for your cryptoHashing algorithm, and those could end up being in the same folder, which doesn't make a lot of logical sense. With this methodology, it's very easy to isolate a particular set of tests. Since they're all in the same folder, it's easy to select them all,

commit them to your repository all at once, etc. And, finally, if you consider this an advantage is that you can name your test anything if they're in that tests folder. So if you don't like naming your files. test or. spec, personally I find this makes it a bit more clear, you don't have to. But what about the other side, tests alongside components? So when you do this, which files are components and which files are tests isn't as obvious. You have to read the full file name to get a good idea. However, the tests are always directly adjacent to the files they apply to. So you never have to go looking at all to find a connected test and component. With this methodology, unrelated tests aren't likely to be in the same folder. If you have two components in the same folder, for example, the cryptoHash and the cryptoUnhash algorithm, then if those tests were to also be in that folder, that makes sense since they're related to each other. However, you have to remember to name your tests properly. You don't have the flexibility in the first method of naming your tests whatever you'd like. But as we'll see, we can use Jest to run tests by name. So even though your tests may be in different folders, you can still run them all at the same time. If you're eager to try this, we will do so very soon. But, first, we need to discuss Jest globals. We'll do so in the next clip.

Jest Globals

So before we create test files, we have to understand these things called Jest globals. So there's a whole variety of Jest globals, but in this clip, we'll be talking about the two most important ones, describe and it. So the it global, which is also known as the test global, is a method that you pass a function to as an argument, and whatever that function is, it will be run as though it's a test by Jest. So this is kind of how you pass your test logic to Jest using it or test. The two methods are equivalent, but for historical reasons, some people prefer to say describe and it, and other people prefer to say suite and test to talk about the same thing. Speaking of suite and test, the other global is called describe, or you might also refer to it as suite. And this is an optional method for grouping it blocks together. There's no requirement that you use describe, though personally I frequently use it in my tests. So if this looks a little intimidating, don't worry. In the next clip, we will be adding these to our application.

Jest Globals - Demo

Dust off that keyboard, ladies and gentlemen, it's demo time. In this demo, we'll be creating some test files. We'll create a test file in a __tests__ directory and make sure Jest recognizes that. And then we'll also create a test file with. spec as part of the file name and see that that works just the

same way. Inside these, we'll put describe and it blocks and see what the interaction is between Jest and these blocks. By the end of this, we'll know for sure that Jest is going to see these files if we follow the conventions that we've discussed. And the files themselves aren't going to actually be testing the logic of our React application. We'll be adding that content to the tests in a later module. So for now, we're just going to be setting them up. They will be very general for the time being. So here I am inside my text editor. I can pop open my terminal here. And typing jest confirms that there are no tests. Let's start by making a __tests__ directory. So I know we're going to want to test our components. I mean that is part of the course name. So let's go to src and components. And inside components, we can add a new folder and call it __tests__. You might also call that a dunder, as in dunder, tests, dunder, if you're familiar with Python. So we'll make this new directory and call it __tests__. Looking good. And now inside, we'll add a file and call it---it'll be tests for our QuestionDetail component. So we'll say new file and call it QuestionDetail. js. So we have a file. It's in a __tests__ directory. Is this going to change what Jest does when it runs? Isn't that cool! So Jest sees that our QuestionDetail file now exists in the __tests__ folder. It runs it, and then it fails because it says, Your test suite must contain at least one test. What a wonderful lead-in to our using describe and it blocks, don't you think? So Jest sees our file, but we're getting an error. Let's fix that. So Jest will throw an error if there's not at least one it statement in an entire test file. Jest assumes for better or for worse that something must be wrong. So we'll start with describe. And the first argument to describe is a string which explains what it is we're testing. Usually the name of the component is just fine. So we'll say describe('The Question Detail'). Let's say Detail Component, so we're now mixing it up with any service or anything. Naming tests is, of course, a fine art. Now the second argument is a method, and this method acts as the scope for all of our tests. Now that we have the describe block, let's run Jest and see what happens. We get the same error, Your test suite must contain at least one test. Well, when we wrote describe, we have a test suite, but we don't have any tests until we use at least one it or test statement. Throughout this entire course, we're going to be using it since that's the more common methodology in my experience. So inside our describe block, we'll say it, and the first argument is the thing that's specific that we're testing. By convention, these usually begin with the words should, like it should work or it should balance the checkbook, but you can come up with your own conventions. Or better yet, use whatever conventions are already in place in your workplace. So I'll make a test, and I'll call it it('Should not regress'). Now we have an empty it block. Do you think that that will be okay by Jest? Yes. So now we're passing. In our it block, we don't make any assertions, but to Jest, a lack of assertions is equivalent to all the assertions being correct. So we'll leave this file by now knowing that we'll revisit it in an upcoming module. Let's make a test file using. spec. So also in the components folder, I'll make a new file and call it

QuestionList. spec. js. So now we have our. spec file. I'm going to run Jest. What do you suppose will happen? Yes, you may have easily guessed that Jest will fail again because we just made a test file with no test. Let's fix this by adding describe and it blocks. So we'll describe("The question list"). And our first test will try to verify that it displays a list of items--it ("should display a list of items"). And you can see here if you put a space after the it and then start the string with a lowercase s, it kind of reads like a sentence a bit. I think this was the idea when they named the method it. Of course, you may think that it doesn't look like a sentence at all and all this describe/it business is quite silly. Well, for very serious people, there is test. In order for suite and test to be recognized, we have to enable TDD mode. We won't be doing that now as calling it describe and it is more than sufficient for our purposes. So we've written two test files, Jest sees them, it runs them, we get no errors. Terrific! Join us in the next clip as we continue to learn about Jest.

Watching for Changes

As we mentioned previously, continuous integration is more likely to run tests, whereas developers are more likely to watch tests. Let's learn about doing the latter, watching tests. So you watch tests by entering watch mode, a special mode where tests are run based upon which files have changed. Jest is smart enough to know which tests relate to changed files and runs those but doesn't run any tests that wouldn't have changed at all. The developer does not need to worry about telling Jest when to run. It's there detecting changes for you and presumably helping steer you in the right direction when you make a change that causes something to fail in some way. This is one of the few tools that we have to actively prevent regression. So if every developer is watching tests all the time as they write code, then you really are confident that you have at least something in place to prevent your application from going back to a state you don't want it to go. So we're going to have a little demo. We'll initialize Jest, and we'll give it the watch flag so it goes into watch mode. And we'll take note how it runs new tests each time our underlying files are updated. We'll also note that we don't have to do anything special to tell Jest the files have changed. It just does its thing. Alright, let's go to the IDE. So here I am in my IDE. As you can see, I can type jest --watch. Now rather than running my tests and completing, it has run my tests and then entered into watch mode. As you can see, it gives you this selection of options that you can actually type right in to your terminal. So if I press the a button to run all tests, it runs all my tests. So now we're in watch mode. What happens if we update this test? I've got the question list specifications open, so I'll just add an assertion here. This is going to be the first assertion we've seen. So if it looks a little alien, don't worry. We've got a whole module coming up where we

discuss these more. So we'll make a very simple assertion. We'll expect(2 + 2). toEqual(5). And badabing, badaboom! The moment I press Save, Jest reran my tests and said, You know what? 2+2 is equal to 4. This is a failing test. So without having to do anything, Jest alerted me that something is failing deep down in my test suite. This also works the other way where if I have tests for a component, and I edit the component causing the test to fail rather than editing the test file directly here, I still get this handy warning. So let's fix this up. Let's just change it to expect(40 + 2). toEqual(42). Our tests pass, and bam! So it goes without saying that using the watch mode is a paramount feature of any closely integrated team that is using tests. Join us in the next clip where we discuss setup and teardown globals.

Setup and Teardown

So we've already learned about the globals describe and it. Globals are so called because they kind of just exist in this magically scope in our tests. You'll notice we just typed describe. There's no explanation as to where this method describe came from or how it was defined. It's just been added to the global scope. And there are a few more automated tools that we can use to help as well. So the before family of globals comprises BeforeEach and BeforeAll. BeforeEach runs a little bit of code before each and every one of your tests. It's useful for any setup activities, so if you want to set up a mock database that all your tests are going to run their tests against, this is a great time to do so. BeforeAll works like BeforeEach, but it only runs once ever. So if rather than having an individual database, mock database instance for each of your tests, just one global one is sufficient. Then BeforeAll is good. On the flip side, we have AfterEach and AfterAll. And these work---they're just the opposite of BeforeEach and BeforeAll. So AfterEach runs a block of code at the end of each it statement, and AfterAll runs it just once at the end of the last one. It's useful if you have a connection that's open, you can close it, or if you've opened a process that you need to terminate, this is a great time to do so.

Setup and Teardown - Demo

Let's demonstrate the setup and teardown globals. So we'll take an existing test that we have and add before and after blocks to it. We'll take note of when these blocks are executed and how many times. Ultimately, we'll be using beforeAll to facilitate the mocking of our application, though we're not going to do that until the module on mocking. But we will put these blocks in and see what happens right now. So here I am in my IDE. I already have Jest watch running, which is convenient because we're going to want to run tests based on our changes anyway. So

first I'll say beforeEach. Pass a method there, and rather than trying to set up anything fancy, I'll just make a console log statement and say Before each! If we look at the output from Jest, we can see here's our console log, and it does indeed come right before our passing of the QuestionDetail spec and right after the QuestionList. spec. So even though this beforeEach came at the beginning, these console log statements are actually only released after the synchronous tests are run. Let's add a beforeAll block. And, sure enough, beforeAll runs and then beforeEach. Good to know that even though we put them in this order, beforeAll is going to run before beforeEach. Now what happens if we have more than one test? Let's copy and paste this test. We'll just give it a nonsense name that's different. Now what do you think will happen when we run the tests? As you may have guessed, beforeAll runs once, and beforeEach runs twice. So it's running once for each of these tests. When demonstrated in this manner, they really do seem quite simple. Let's wrap up by using afterEach and afterAll. I'll just type them both up now. And I'll make a point of putting the actual code before my it statement. So I've added afterEach and afterAll, and if we see what happens with our code here, very cool. So, first, we see Before all! Then our first test runs. Before each! runs. It finishes. Then After each! runs. Then our next test starts, and we again see Before each! and After each! And, finally, with all of them complete, we see After all! So in an upcoming clip, we'll be using these to actually facilitate mocking. But for now, it serves well just to see them working. Feel free to kind of play around with the file that we have now. Put different statements in if you're still not sure exactly what's going on, and we'll see you in the next clip when you're ready.

Skipping and Isolating Tests

Skipping or isolating a test is something that developers have to do in practice every day. Skipping a test marks it in such a way that the test is not run when Jest runs tests. This is really useful. For example, you may be in a workplace with a less-than-ideal linkup between your testing and your version control, and someone might commit a test that's just failing. So now you're trying to work, but everything you do results in tests failing. You don't have time to deal with this, and this person just went on vacation, so that test may not be fixed by the personal who originally messed it up for a few days. So you can just skip that test, and Jest will go back to passing, and you can get back to your work. Isolating a test is like the flip side but it has the same effect. If you're in that same situation where someone else's test is persistently broken for a while, you can rather than skip it isolate the tests that you're interested in. This will have the same net result of the results of the tests that you weren't interested in not being seen. So let's do a quick demo. There's really not a whole heck of a lot to skipping and isolating tests. We'll mark a test as

skip and notice how Jest deals with it differently. And we'll also mark a test as only, which isolates the test. And this will also change the way in which Jest interprets our code. It should be pretty straightforward, so let's jump into it. So here I am in my __tests__ file. Let's start off with only. So we'll say it. only("should display a list of items"). And we'll skip the meaning of life test. As you can see, Jest indicates that two of our test suites have passed, but one of our tests has been skipped. As you can see, if we change our skipped test to something that would fail, for example, 40 + 2 = 43, Jest does not fail because if skipped tests fail, then the test runner still exits successfully. If we stop isolating this test by getting rid of only here, one test now fails. On the flip side, we can say it. skip for our now-failing test, and it works. We can give them both only here, and this results in everything being labelled only to run, so this is equivalent to neither of them having anything. And that is how to isolate and skip tests. Let's finish this off by un-isolating this test but keeping this test skipped. This results in all our tests passing, and we're looking good.

Async Testing

Finally, let us discuss and demonstrate asynchronous testing. Asynchronous tests are something that one must do all the time while making real-world applications. And it's also something that Jest does especially well. So an asynchronous test is just like a normal test. It contains assertions like toEqual or any other assertions you would like to use. However, unlike the tests we just saw, an asynchronous test does not complete right away. Sometimes it could take a long time. Sometimes it could take a short time. Heck, sometimes you don't even know how long it's going to take because it's testing using dependencies that you have no control over. So in all of these cases, there needs to be some way of notifying Jest that the test you're trying to do has concluded. So how do we notify Jest? Well, there're a few different ways to define an asynchronous test while testing with Jest. You can request that a done callback be passed to your test and then invoke it when the specified amount of time has passed. You can also return a promise from a test. Any test that returns a promise will be seen as asynchronous and won't be complete until the returned promise resolves. Very cool! Lastly, and this is the coolest one, you can pass an async function to describe, and you can handle the asynchronicity in the very way you would do so using an async function, which is by far the cleanest syntax. Let's have a peek. So here at the top, you can see a syntax example where we're using the done callback to finish the test after 100 msec. In the second example, there is no done callback, but we're returning a promise. And in the third example, we're not returning a promise but, rather, the function that's passed is an async function. Each one of these three tests do roughly the same thing, so you can

see it's really about style. And in case you're wondering, delay is a method that returns a promise. It's a utility that comes with redux-saga.

Async Testing - Demo

Welcome to the concluding demo of this module. So in this module, we're going to write some asynchronous tests. We're going to write one with a callback. Then we'll take special note of what happens if the callback is never called. And we'll also be writing a test with a promise and one with async/await. These tests will be fairly abstract, so they won't actually be testing the functionality of our application yet, but we'll use these formats of async tests in the future once we understand them. So here I am in my IDE, and I'll make a new file just contain these asynchronous examples. I'll call it App. spec. js and put it next to App. jsx. So let's have some fun and write some async tests. Let's start by doing the simplest one, one you're familiar with if you've basically written any tests in any language, the callback method. So we'll say it, and we'll just call it async test. And the method that we pass to it has one argument, done. Note here how we're kind of combining the describe and it syntaxes where this time we haven't wrapped our it blocks in a describe statement. And then rather than using should as in it should do something, we're just naming our it blocks like they were describe blocks, it("async test 1"). So it's not the way I prefer, but it's another way of organizing your tests. So all we'll do here is we'll just say set a timeout, and then we'll call done after a short period of time, 100 msec or so. Now let's pop open our terminal, and I'll start Jest, but I'll say jest app. This results in only App. spec. js running. Anything that you put after the word Jest right from the command line becomes a regex, so it'll search for any test files that matched that regular expression. Very cool! Thanks Jest! So let's actually run that again with watch, so we'll say jest app --watch. I'll just move my terminal there to the right so we can see it a bit more tall. Now let's just try changing this timeout to 1000. What's going to happen? That's right. So the test still runs, but it takes its sweet time in doing so. What if we make it a really big number like 6000? We actually get an error because the maximum callback is 5000. You can change a setting to fix this, but this is generally a good default, so if you have to make the timeout longer than 6000 or 5000, maybe you're not testing the right things. Maybe your application just isn't right. And what happens if we just don't have this line here? What if done is just never called? Well, sure enough, it waits for the callback for the minimum timeout. Done isn't called, so it errors out in exactly the same way. Let's fix this by making done called after 100 msec. Now it works. Next, let's make another test, async test 2. This one doesn't have done as an argument, but rather it returns a promise. So we'll return a new promise. And we'll pass it one argument, a method taking its own argument called resolve, and we'll just set a

timeout and call resolve after 100 msec. And as we can see, that works the same way. We can stagger them, so I'll make this top one 100 and this bottom one 1500. Save it to run our tests. And you can see here it notifies us that async test 1 took 100 msec, give or take, and async test 2 took 1500 msec, exactly as we'd expect. Finally, let's try async/await. So we'll write async test 3. Now unlike the top two, this one takes an async function, which if you've never seen, that's okay because it's a very new syntax. And we're just going to do it right now. So we'll say async, and then we'll define a method right next to it. So the way the async methods work is you can use a special word within them called await. And after await, you have to give it a promise, and the code won't resume running until the promise resolves. So I'll actually import delay from redux-saga since it's my favorite utility for creating a promise that resolves after a certain amount of time. And now here we can say await delay(100). Redux-sage is, of course, available because it's part of the underlying isomorphic React application. And, indeed, the test passes once again. The test passes instantly, which suggests that redux-saga is interacting with Jest in some way and replacing a longer delay of 100 with a shortened instant one. And that's all there is to it. We'll be using many of these asynchronous styles in upcoming chapters to write some actual tests for the application. But that is how it's done. In the next clip, we'll wrap up this very education-filled module.

Summary

That was no simple module, folks. If you're anything like me, you probably need a 5-minute break to process all the stuff we just learned. But, first, let's kind of summarize everything that just happened in that module. So we learned that we installed Jest, we watch our tests, and we run Jest through the command line and the related Jest CLI library. We learned that for continuous integration, you mostly want to run tests, and for a human developer, you mostly want to watch tests. We learned two different ways of naming tests, putting them in a __tests__ folder or giving it a spec or test extension. And we also learned about global configuration with before and after in those globals. We also learned about skipping and isolating tests. And we saw there's not one, not two, but three ways of writing an asynchronous test in Jest. Coming up in the next module, What are mocks? Are they things? What do they do? Let's find out. We'll learn about the require global, a fiendishly complicated global whose complexity far exceeds those of the primitive before and after globals we've seen so far. We'll learn about mocking different kinds of modules and what that involves. We'll also play the spy game and learn about observing functions and the arguments that are passed to them. Finally, we'll learn about automatic and manual mocking, the difference may surprise you. All this and more in the next module. I'll see you there.

Mocking Functions and Modules

Why Mocking?

Welcome to this module entitled Understanding Jest Mocks. Mocking is a big part of what Jest does. So unless you understand it, you can't get the most out of your Jest setup. As Laurence Olivier said, We ape, we mimic, and we mock. " I don't know how much aping we're going to do, but in the sense that they both involve mocking, Jest is a lot like the theater. So when we mock something, we're necessarily going to have to do a bit more work, so the question is, Why should we mock something in the first place? Are there any advantages to doing so? So, first of all, mocking reduces the number of dependencies, the number of related files that have to be loaded and parsed when tests are run. So using lots of mocks just results in faster test execution. It only saves a little bit of time every time, but over thousands of repetitions, it can really add up. Additionally, if your components aren't built from the beginning with testing in mind, they can easily be built in such a way that calls to actual APIs, maybe even vendors, are hardcoded in and hard to avoid when writing tests. But by mocking their dependent modules, we can prevent these side effects without even needing to change the file itself. Finally, we may have in mind a very specific test for a component that, say, gets its state from many other different components. To set up this perfect scenario to get the component in the state where we want to do our test, we can use mocks to get the data that we need to the component. So now we know the Why. In the next clip, we'll discuss What. What is a mock?

What Is Mocking?

So what is a mock? Let's start with a simple example. Let's say you're sitting at your desk at work writing some code minding your own business. All of a sudden, a scientist comes along and decides it would be fun to create a clone of you. The clone of you is in many ways like the mock of a module. The real you has your appearance, but you also have a real purpose. You're there to do something. You have feelings or, in other words, you have complex internals that aren't visible from the outside. However, your clone is a bit different. Appearance-wise, he looks just like you. No one can tell you apart. But your clone doesn't have any inner workings, any feelings. Inside, he's hollow. In fact, the only reason the clone exists is to convince other people that he's you. So I pose the question again, What is a mock? So a mock is basically a convincing duplicate of an

object or a module without any real internal workings. It might have a tiny bit, but compared to the thing that it's mocking, it's incredibly simplistic. It can be created through an automatic process as facilitated by Jest or created manually, which is a bit more work but gives us more control on how our mock behaves. The modules in your application interact with the mock in the exact same way they interact with the real module. In fact, from their perspective, there shouldn't even appear to be any difference. However, one thing that is notably absent is side effects. Since the mock has no real internal workings, rather than actually calling your vendors API, it simply does nothing or pretends to. And then we can add to that more advanced mocking features such as spies, which we'll be discussing, which add even more functionality to our mocks and make them better clones of the modules that we're trying to test. In the next clip, we'll discuss the mocking process.

The Mocking Process

So how does one make a mock? If you're the scientist in the previous example, how do you create an unthinking, unfeeling version of a module? In fact, the process for doing it manually and the process for doing it in an automated way using software are very similar. Other objects usually interact with a module through its API methods. That's like the doorway to get into the object. So the clone needs methods of all the same names. For these methods, we use spies, which as we learn are like methods but provide lots of additional metadata. In addition, one of the caveats of mocking is that if you mock a function that returns a promise, but the mock itself does not honor this promise return, your code will throw an error when it calls. then on something that's undefined. So more than just creating a stub for every function, you have to make sure the return values make basically as much sense as they have to. If you have something that returns a promise, then returns a promise, then returns a promise, this will still result in an error if your code is written in such a way to expect that. Things can sometimes get tricky when this happens, and it's still a great unsolved problem in computing how we can mock methods like this. Lastly, if a method were to return something complex, say, an object with a nested object with a nested object with a property we were interested in, the object would have to match the shape of the original. Otherwise, just like with the previous example, when attempting to access a property of an object, which is undefined, the code will simply throw an error, and you'll be unable to determine if it works or not. In the next clip, we will talk about spies.

Mock Functions

Let's discuss mock functions. Mock functions are also known as spies so called because they give you insider information on how the method is being interacted with. Mock functions don't create any side effects. In other words, the side effects that may be created by the functions that are mocking are notably absent in the mocked versions. Mock functions or more specifically spies, since when we say mock function, we could be referring to a function that doesn't use the spy API but, rather, something that we just wrote ourselves, when we call it a spy, it's very clear that it has this additional functionality such as counting function calls or every time the method is called, it keeps track of it so you can easily run assertions against this. It also records what arguments are passed in so you can easily verify by mocking a dependent service that a component you're trying to test is calling it in the right way. You can load your mock function up with return values. So if, for example, one of your components calls a user information service, gets some information back, and then displays it in a certain way, you can load a mock of the user information service to return specific information that will result in the component being rendered in a certain way, a way that you want to test. And as we discussed, if the original returns a promise or a promise that returns a promise, etc., the new value has to be more or less like the old value, or you're just going to run into errors. It's very likely. So we will shortly be getting into some hands-on coding as we write some mocks. But, first, let's talk about the actual process for creating mock files, which we will do in the next clip.

Creating Mock Files

So how does one actually create mock files and tell Jest to use them? So if you name your mock file appropriately and are mocking an npm module, then it's loaded automatically for all of your tests. The default behavior of local modules is different, and you must specify in your test what you would like to mock and which mocks you would like to use. To be appropriately named, a mock must reside in the __mocks__ folder. And that folder needs to be in the same folder as the module that you're trying to mock. Lastly, both npm modules and local modules can be mocked, though the process differs slightly as the dependencies are handled differently. In the next clip, we will be doing a code-along demonstration of creating mocks. So, get that text editor open, and I'll see you there.

Mocking - Demo

In this demo, we'll be writing a group of tests for our question-fetching saga, which as you'll recall responds to action and generates an API call, in other words, a side effect, before returning the

appropriate data for inclusion in the Redux store. So we're going to create a mock of the isomorphic-fetch npm package. This is a very common and simple package which allows you to make Get and Post requests in a manner that's easily adapted to both the browser and a Node-like environment. We're also going to add some custom functionality to our implementation of isomorphic-fetch so we can preload whatever value we want to create the testing scenario we're interested in. We'll take note that since this is a global mock, it will be automatically substituted in for any tests as soon as it exists. In this clip, we won't be mocking any locally defined components, but we will do so in the chapter on testing React components since I just thought it fit so well in there. Alright, let's head to our text editor. So here I am in my code base, and we'll be testing the fetch-question-saga. Let's have a peek at this saga. So basically what it does is it responds to every REQUEST_FETCH_QUESTION, and then this block of code runs, which calls fetch from isomorphic-fetch and actually calls the Stack Overflow API. Now I should mention that if I had used yield call fetch, which is a more sophisticated way of doing some of the redux-saga, this would actually work without any mocking. It wouldn't even need to mock the isomorphic-fetch package. That's actually the better way of doing it. But the way I did it here is it actually makes it convenient for teaching the lesson on mocking modules. So that's fine. So as we can see, if we called handleFetchQuestion without mocking fetch, it would actually call Stack Overflow. If we're paying money every time this happens, this is an expensive problem. So we're going to want to test this specific generator function. So usually it's a good pattern if we can test the smallest part of our program possible. So you may think that it makes sense to test this default function and then create some kind of setup where we're dispatching actions so this triggers, but really that's not necessary. First, what we'll do is we'll export handleFetchQuestion. And as you can see, it's actually just a simple generator function, which takes an argument and then predictably generates a series of side effects followed by an action. So let's create a new file right next to fetch-question-saga, and we'll call it fetch-question-saga. spec. js. And we'll start by importing handleFetchQuestion, the method which we just exported. And we'll write a describe block and an it block, and we'll just call those ("Fetch questions saga") and it("should fetch the questions") respectively. Of course, you can name it what you like. Let's make sure everything is working so far, so I'll put a console. log here. Now let's open up our terminal. And since we only want to run this test, we can just say jest and then type in anything that would be a regex for this file. So I can say question, jest question. And, sure enough, it only runs---well, actually, it runs this, and it also managed to pick up QuestionDetail. js, which I was not expecting, but that's the correct functionality. Let's actually say jest fetch-question. And we see it runs just the test we specified, and we don't get errors, and we see our log. Good, we can now proceed. So what we'll do here is we'll actually make this an async function, which will let us await values from the

generator. Now we'll create a new instance of the generator by saying const gen, and that'll be equal to handleFetchQuestion, and we'll pass in a question_id of, let's say, 42. Get rid of this log. Now this just creates the generator. To make it run, we have to call gen. next. This is just how ES6 generators work. So we'll say const value in square brackets, that should be pointy brackets. And it'll be = await gen. next. This should be interesting. What happens if we run this test as it is? Fascinating! So as you can see, we get a truly puzzling error, only absolute URLs are supported. With a little bit of inspection, we can see that this is the error that isomorphic-fetch throws if the URL is anything that it doesn't like. Puzzlingly, this error is not thrown either in the client or server versions of the code. But when trying to invoke the real handleFetchQuestion in our test, this happens. Obviously we can't have errors like this appearing in our output. If we try to commit this code, our senior developer would surely have us starkly reprimanded. So our goal now is to make this error go away, and in the process, maybe make our isomorphic-fetch page a bit more amenable to this test. So what we'll do is we'll create a new folder called mocks. Now the mocks folder has to be next to whatever is being mocked, and the mocks that're meant to serve the node_modules folder have a few special rules. But suffice it to say, it goes at the top level of your application next to node_modules. We'll call it __mocks__. Now if a file in your __mocks__ folder that is directly adjacent to your npm modules has the exact same name as an npm module, it will be loaded instead of the whole module when you require it in your tests. It's beautifully simple and just a little bit magically. Let's give it a shot. So we want to mock isomorphic-fetch, so we'll make a new file and call it isomorphic-fetch. js. So the way isomorphic-fetch, our mock, will work is it has the same API, but it only has one mock value. That value is specified by something that requires it, and it'll always return that. So let's start by defining the value, and we'll just call it __value, and we'll make that equal to, you guessed it, 42. Now we'll define the method. We'll say const isomorphicFetch =, and we'll say jest. fn. So this creates a spy function. So if we just called it like this, it would create kind of a dumb function which doesn't do anything, doesn't return anything, but still keeps up information about what is calling it. However, we can pass a function in, for example, this function that takes no argument, and returns __value. And then that function will be wrapped in a spy. So the value here, isomorphicFetch, is now a spy which always returns __value. That's pretty handy. Finally, we'll define an API to set this value. So we'll say isomorphicFetch. __setValue, and we'll make that equal to a method that changes this value property that's here on the global scope of this module. You pass it v, and __value is not equal to v. Finally, we need to export this as a default module. This will let Jest know to replace the module isomorphicFetch with this one that we've just written. Alright, so we've written that. At the very least, we should hope that our code runs a little differently. Let's open our terminal up and run that test again. Alright, the error is gone, but does it actually work? I think it works, but let's make

sure. So we'll go back into our spec file. So we have this value, so we'll just run an assertion. We'll say expect(value). toEqual, and it will probably look something like this, an array with an object, and then the object has a question_id property, and we'll just make that equal to 42. So now if we run this test, it fails. It expected this array type object, but it really got the number 42. Investigating our package, we can see that this is how it's set up. So we need to change the default value. We need to load it with something so our test will be happy. This is a perfect occasion to use beforeAll. So we'll call beforeAll, and we'll say---well, first, we need to have fetch in the scope. So we'll say import fetch from 'isomorphic-fetch'. And note that this is automatically going to be the mocked version. We can establish this because our mocked version has the __setValue property. So we'll say fetch. __setValue and pass it in an array with an object question_id:43. See what I did there? This is going to fail again, but it'll fail differently. Let's try it. So we get a different error here. It wants the question ID to be 42, but it's 43. So let's now use our setValue API to get it just right so our test has the correct value. We'll save it, run, and it passes. Cool! But, wait, there's more. Finally, since fetch is a spy, we can actually use it to do more assertions. We can expect(fetch), and then we can use special spy matchers. The one we'll use here is toHaveBeenCalledWith. Wow, that's a mouthful. So what this method does is we pass it a string, and that string has to match something that it was really called with. So here's a great opportunity. Let's type in what the URL should look like. Let's say /api/questions/47 and run the code. And it tells us clearly that it was never called with this string, but it was called with /api/questions/42. I love it. I love a good spy function. Let's fix that to 42, run our test, and everything's passing. That's pretty cool! So in that example, we did quite a bit of mocking. And, well, it was challenging, but I sure hope we all learned a lot. Coming up, we'll wrap up this module, as well as have a special discussion about automatic and manual mocking.

Automatic and Manual Mocking

In the previous clip, we saw how it's possible to mock a component manually. Jest also has the capability of generating mocks automatically. Though I usually prefer to mock things manually myself, let's discuss both these capabilities. In some setups, depending on the configuration that you have set up and the version of Jest you're using, any required modules may automatically have mocks generated for them. We saw in the setup that we have going on here, that's not the case. And there was no mock generated until we used a manual mock that we created. However, we did see a hybrid of this behavior, as once we wrote a manual mock for isomorphic-fetch, we didn't have to specify use it. It took that and automatically put our manual mock in for any module that required it. We can easily see that most applications require some combination of

this manual we-write-the-code-ourselves mocking and this automatic or magical mocking that seems to all happen behind the scenes. So the properties of manual mocks are as follows. They exist as a separate file alongside the file we're mocking just like the one we wrote. These manual mocks once they're in the right place will be used automatically for npm modules, though this is not the case for local modules. Writing mocks is more work than just having Jest generate the default automatic mock. And if we've written a mock for a package, but then that package's API changes, the mock will also need to change, or it will no longer keep up. And here are the qualities of automatic mocking. Most modules can be replaced using an automatic generate mock from module system. This usually works, but sometimes is doesn't work perfectly. Generally speaking, you can't assume your automatic mock will work. Often you need to add more functionality to it so that it closely enough approximates the original so that components that require it won't throw any errors. However, just having modules mocked automatically or using mocking as much as possible does greatly reduce the odds of side effects, which is a great consequence to have for using this system. As always, the developer must use their head. Sometimes automatic mocking makes sense, and sometimes it doesn't. Here are some challenges that you'll face if you try to implement automatic mocking. So methods that return a complex value can't easily be mocked. What if a method returns an instance of a service which itself is a factory, which returns an instance of an API request which resolves to a value? Admittedly, you should have these broken up so you can test them piece by piece, but assuming you can't, mocking this is really complicated. And then what if it changes after you've written the mock? Clearly this is not a very efficient way to go about working. In addition, JavaScript itself is a pretty tricky language. It has many other features that would be considered, well, pretty heretical in other languages like the ability to add a new method to your class simply by, well, just adding a property to that prototype's object. You can prove mathematically that there's no algorithmic way to stay ahead of all the trickiness a clever JavaScript programmer can throw at you. So there're lots of ways that methods that are there that you expect to be tested simply won't be mocked. And, lastly, depending on your configuration, modules that you did not expect to mock may be mocked, which can lead to unexpected errors that can take all day to figure out. Historically, Jest automatically mocked more modules than it did now. However, due to this problem where modules that you did not expect to be mocked were mocked, the behavior has changed to the current functionality, which is that generally we have to at least write a file in the appropriate location for a mock to be used. In the next clip, we'll wrap up what we've learned in this module.

Summary

So what did we learn in this module? We learned that a mock is a convincing duplicate of a module or function. It has no inner functionality, but the API can fool most consumers. We learned that simple modules with very few APIs and primitive return values can be automatically mocked, but more complex ones will need manual mocking, and this is generally the kind of mocking that you'll be using. Finally, we learned that you can specify a file is a manual mock of something just by giving it the correct name and putting it the __mocks__ directory. Coming up in the next module, say cheese as we explain snapshot testing. We'll look at the big picture and tell you what the advantages and disadvantages are of snapshot testing. We'll even learn how to update snapshots, something that as a developer you'll be able to do on the fly. Overall, writing the module on snapshot testing is one of the things that got me most excited about this course, and I'm sure that it's the main attraction or one of the main ones for you too. So stick around, and we'll jump right into it.

Snapshot Testing

What Is a Snapshot?

Hello and welcome to this module titled Snapshot Testing. I am simply so excited to jump in to this module. Snapshot testing is just such an important part of Jest. I'm sure that coming into this course, it's one of the parts about Jest that you've heard about and wanted most to learn about. Your enthusiasm will soon be rewarded as we're about to see how simple and useful snapshot testing is. "A good snapshot stops a moment from running away, " - Eudora Welty. This quote quoted from a person who ostensibly did not know much about code or tests really summarizes the principles behind snapshot testing very well. In theory, we're talking about a moment in time when your component or the component worked. If you use snapshot tests properly, you won't find yourself in a situation where your component suddenly doesn't work. But let us pause to consider an important question, What is a snapshot? So the actual code representation of a snapshot is a JSON file, and this JSON basically contains a record of what your component looked like when the snapshot was made. During tests, Jest compares the contents of this JSON file to what is actually output by the component during the test. If everything matches, the test passes. Otherwise, it fails. One interesting note is that these test snapshot files are actually committed to the repository along with your actual modules, tests, and files. We'll discuss the

advantages and disadvantages of this unique feature in a later clip. In the next clip, we'll discuss how snapshot testing works.

How Snapshot Testing Works

Here's how snapshot testing works. The code above represents basically the complete code required to make a snapshot test. First, the component is imported, as well as the renderer from react-test-renderer. We use the renderer's. create method to create something called a tree. The tree is basically a representation of the HTML output of the component. Note how we've used react-test-renderer in this example. We've made a choice due to the large number of open issues currently on its GitHub repository to just not use Enzyme for this course despite the fact that it does have certain advantages. Once you have this tree, all you have to do is run the assertion to match snapshot. So to match snapshot is an interesting assertion in that it works differently the first time it's run than each time subsequently. It would be hard to imagine an assertion like toEqual or toBeGreaterThan to work differently the first time it ran than the second time given the same numbers. However, the first time you call it, it just creates the snapshot. It has nothing to compare it to, so the test automatically passes. Essentially the first time you run a snapshot test, nothing is even being tested. There're no circumstances under which it can fail unless you just have a huge problem with the component itself. But once the snapshot file which as we learned is a JSON file exists, the snapshot will be compared with that JSON. If it doesn't match, the test will fail, and the JSON will have to be updated, which we have a whole clip about coming up. To bring it in to more of a workplace perspective, here's the snapshot testing process as it would be used in a common style team delivering some kind of React application. Developer A, who you can imagine as yourself, creates a new React component. They add a snapshot test, and the snapshot of that component is generated automatically by Jest. Having done a good job, developer A commits their changes including the snapshot test and moves on to another project. Developer B at this time is working on something else, but it's a dependency of the component. As the final part of the update process, they run the Jest test suite. A new snapshot is now generated, but then rather than replacing the JSON file, it's compared to the existing file. Since they don't match, the test suite will fail, and Developer B will know to backtrack, determine the problem, and fix their update so that the component doesn't change. Or if the change was intended, they can update the snapshot test. In the next clip, we'll be coding up our first snapshot test in the application. It should be great, so I'll see you there.

Snapshot Testing Demo

Aren't demos just the best? I sure do love demos. In this demo, we'll be adding a snapshot test to a component. As we just saw, there's not a whole lot to adding a snapshot test. It's very simple. So this won't be a very long demo, but it should still have a lot of practical applications. After we do that, we'll be able to note ourselves how the test will fail on even the tiniest change to the component after the first snapshot is generated. So let's go to our text editor. So here I am in my text editor, and we're about to get started. But, first, let me show you the application quickly. We'll be updating the TagsList component, which just shows all the tags related to a question. It's a very simple component that just essentially is your basic repeater type of element. So let's make a snapshot test for it. So let's go to the src/components folder where our components reside. And we'll make a test file for the TagsList. I'm going to name mine TagsList. spec in the same level as our component. But if you wanted to put it in the __tests__ directory, well, you could. I'm not going to stop you. So we'll name that TagsList. spec. js, and the name if it's not in the __tests__ folder is important because that's how Jest is going to recognize this test. So first let's do some imports. First, we'd import React. Anywhere you use JSX, you need to have this React import statement for technical reasons. We'll be using a bit of JSX. Hence, we need this import statement. Next, we'll need to import the component itself, so we'll just import TagsList from TagsList. As we can see by looking at TagsList, this is not a connected component or even a stateful component. This is basically the most simple kind of component you can get in React. But, hey, it still does some pretty cool stuff. So back in our spec file, finally, we'll have to import renderer from 'react-test-renderer'. We'll also need to install this dependency, so let's open up our terminal. So we'll say npm install --save react-test-renderer@16. 2. 0. Now 16. 2. 0 isn't necessarily the newest version at the time that you're watching this. It's the newest version at the time this course was made in April of 2018. So you should still use this version for this demo so you're not confused, but if you go into a workplace, and you find a version that's much earlier or much later, you'll need to be able to adapt. Luckily, 95% of what we learn about this version will apply to any other version. So let's write a test block. We'll describe("The tags list"). And we'll try to verify that it("renders as expected"). So, first, we'll define our tree, which is a JSON representation of our component. More specifically, it can be JSON, but you might also find a tree with a different structure like HTML. So we'll say const tree, and we'll make that equal to renderer. create. So we pass a React component to. create, and what renderer returns is the generated HTML. No DOM required. So we'll make a React component with these HTML-style brackets, and we'll say TagsList. And the TagsList takes a tags prop, which is a list of strings which are the tags. So I'll just type some in. You can type whatever you want. And put these in curly brackets so React knows it's an array and not a string that looks like an array. So I'll give it a couple of elements--CSS, HTML, and maybe a little go. So that's our tree. And, lastly, let's just call toJSON. So now the tree

represents a JSON version of the component. Let's pause right here and log our tree so we can actually see the tree that's being rendered. We'll say console. log(tree). Now open up my terminal, and I'll say jest, and do you know how we can get Jest just to test this file? That's correct. Let's put the word TagsList after jest so it runs a regex. So as you can see, it's logging the tree that's generated. The tree doesn't look much like HTML. You can see it's storing all these child HTML nodes as kind of detailed metadata-ful JavaScript objects. This is all the information our console will give us, so let's go back. I'm just going to close my terminal. So now we'll say expect(tree). toMatchSnapshot. If you can believe it, that's it. That's the whole snapshot test. I told you they were simple, right? So let's run our test again. And it passes. In addition, we get this handy message--1 snapshot written in 1 test suite. Snapshot written, eh? What could that mean? Well, if we have a look in our components folder, a new directory, __snapshots__, now exists. And here's our snapshot. Let's have a look. So you can see the actual contents of the file seem just to contain kind of a name for the test, and what would essentially be the output HTML of the component. So once again, now if I run Jest, the first time it wrote the snapshot. Now it just tested the component versus the snapshot. And, of course, everything is okay. Now let us make a tiny change to the TagsList component. I'm just going to change one character. You know what? I'm excited about this, so I want an exclamation mark after every tag. Add that there. So now our application looks like this--javascript! php! That's me saying the words if they have an exclamation mark after them. Or you might read that as html factorial if you're into math. Nonetheless, our application has changed ever so slightly. Will our snapshot test pass again? No, the snapshot test failed. One snapshot test failed in one test suite. Here's the error message. As you can see, it shows us exactly what it's not expecting. All the exclamation marks have been isolated, and this plus next to them is saying, Hey, we noticed this additional thing in your components. So that's how it works. Rather than update the test, which we have a whole other video to do, let's just change this back so it's the way it was, run our test. It passes. How lovely! And that's snapshot testing in a nutshell. So now that we know how to roll up our sleeves and make some snapshot tests, let's discuss the advantages and disadvantages of them. We'll be doing that in the next clip, so stick around please.

Advantages and Disadavantages of Snapshot Testing

So probably if you ask any developer who has a good amount of experience with snapshot testing what they think, they'll tell you snapshot testing is awesome. And, well, it is in my opinion at least. But it is not all advantages. There are tradeoffs that we must make to use snapshot testing. It's important that you know these especially if one day you're in the position where you

have to decide whether or not to use them in a big project. First, what are the advantages of snapshot testing? Well, it's fast and automatic. We just wrote a snapshot test. It barely took any time, and we barely had to know anything to do it. Snapshot testing catches regression really, really well. So even if you have a human being that's looking at your application after every change, they might not catch a change that's so minute that it can only be caught by a snapshot test. As we've discussed a few times, Jest is not only for React. Jest and a snapshot test can be used for almost any framework. It specifically works for libraries that take a state and then out comes HTML. This is the behavior of all our more familiar engines such as React, Angular, and Vue. For other kinds of tests like, say, testing a service or a saga, snapshot tests can still be used, but they are of diminished value, and this isn't how they're intended. So a big advantage of snapshot testing is if you literally don't have time to write tests, and this does happen in a project that has a lot of goals and a tight deadline, at least snapshot testing is something. It takes just two minutes to write, and it provides you some protection against regression. Actually, the protection against regression is pretty good. The only problem is it's almost too good, so even the tiniest change will require fixes to the snapshot test. Finally, like I said, you don't have to know a lot about snapshots or testing or even coding to make snapshot tests. If you're onboarding a new developer, it's much faster to show them how to make a snapshot test for a component than to teach them everything else about Jest, like I just taught you in this course, that you need to know to write a full suite of fully functional tests. However, there are disadvantages. What are those disadvantages? So snapshot tests are really easy to ignore and even easier to suppress. Put yourself in the shoes of an equally busy rushed developer who notices that their change has broken a snapshot. Well, just one keypress away, and the problem is gone. That sure was easy, and now we all get to go to lunch half an hour earlier. Well, not really because the component is still broken, but we've just seen now easy it is to ignore a snapshot test and make it go away. It's a lot harder to just skip a test that was written by a human being as this raises suspicion from senior developers. Next, snapshots only protect against one thing, which is regression. There're many other things out there that you should be testing for. And snapshot testing does not provide protection against these at all. Additionally, snapshots have an interesting property where they will also break if a component goes from broken to fixed. So whether you have a good component that you broke or a broken component that you fixed, both of them will set off alarms in your snapshot tests. Additionally, we mentioned that snapshot tests commit files to your repository. Most people have the opinion that there're already plenty of files in most repositories. Adding more files to it while necessary a lot of the time is never what we want to do. Additionally, snapshot tests are sensitive to changes that are so minor, they don't affect anything. We might call these incidental, which have no effect on business goals or the user experience or any effect

whatsoever. These will still break your snapshot tests, and you'll have to spend the time to fix them. So ultimately we can see that if you make a component, and you know it's going to change really, really soon, then a snapshot test is almost a waste of time. The snapshot test will break as soon as the component is updated. And while updating the snapshot test itself is simple, ultimately if it never protected against regression between when it was written and when the component was changed on purpose, it didn't do anything. So take that with a grain of salt. In the next clip, we'll talk about updating snapshots.

Updating Snapshots

Working with snapshots is a two-step process. You have to be able to create them, but you must also understand what it is to update a snapshot. As the bard once said, "... Nor all the drowsy syrups of the world shall ever medicine thee to that sweet sleep, which thou owedst yesterday. " Isn't that just a lovely quote? In the world of yesterday, our application worked. All our snapshot tests passed. It was wonderful. But today rolls around, someone makes a change, and that spirit of bliss is gone forever. In this sense, updating snapshots is a little like sleeping in. So what are the nuts and bolts of updating snapshots? So basically Jest has a flag, --update, which says that in the event of a failing snapshot, rather than the test failing, it should simply replace the snapshot with the newly generated version. Just like that, the old snapshots are gone, and the new ones which are guaranteed to match the output of the component are in. However, you might have noticed if you did a little thought experiment that if you run --update and all it does is suppress warnings about components that you have legitimately broken, then you're really diminishing the value of snapshots. And, trust me, using the --update flag is the first thing that any developer is going to want to do because it's very fast and easy. However, --update should only be used once the developer has legitimately verified that the change to the broken component is not a regression but, rather, intended. Stick around because coming up, we have a short demo on updating snapshots.

Updating Snapshots Demo

In this demo, we'll be updating snapshots. It's not going to be a long or challenging demo because, as I've mentioned, updating snapshots is really easy. In fact, it's so easy, it is almost a problem. So, first, we'll change our component so that the snapshot fails once again, and we'll note once again that even the tiniest change to a component makes the snapshot test fail. Then we'll use the command line and the --update flag to update our snapshot. And we will be able to

see that the snapshot test no longer fails. That's all there is to it folks, so let's go. So here I am in my text editor. We'll go to our terminal, and we'll run jest TagsList. And we see everything is passing. That's looking good. Now let's modify this test so that it fails. The last time we changed the component to make the test fail, but we can just as easily make it fail by changing the simple detail of the test. So let's say our senior dev came to us, and he said, You know what? I don't really like go. I prefer swift. Could this test have swift in it? Well, anything for you, senior dev. Let's replace go with swift and run the test. So the test fails as expected, and the test is very smart and knows exactly that we got rid of go and added swift, and it's like that doesn't match. In this case, we know for sure that we haven't broken the component. We've just changed the test for stylistic reasons. So this is an appropriate time to run Jest with the --update flag. As we can see right here, we can use -u as a shorthand for --update. So we'll say jest TagsList -u. And badabing, badaboom, the test now passes. It notifies us that the snapshot has been updated. And if we look at the snap, swift is now in the last spot instead of go. So the old snapshot has been undated. Unless we access it in old versions of a Git repo, it's gone and forgotten to all. And with this snapshot test written, we can commit our changes and finish up. Let's wrap up by committing our snapshot just like we would in a real project. If I say git status, you can see here're a bunch of changes from the last few demos. And also we have src/components/__snapshots__. So I'll copy this, the __snapshots__ directory because I just want to commit that. It is very tempting to just commit everything at the same time. I've been there, but let's not do that this time. So we'll just say git add and then paste the path of those snapshots in. It pressed Enter for me automatically. Thank you. I'll press git status again. So our snapshot test has been staged. We'll commit it by adding a message and running Git with the commit command--git commit -m "Updated the snapshot! " And there you have it. So that's snapshot testing. I hope you enjoyed those demos. I thought they went really smoothly. And in the next clip, we'll summarize what we have learned.

Summary

Well that was just a sensational chapter. If you thought so or, better yet, thought of some way I could have improved it, feel free to tweet at me and let me know your thoughts. My Twitter handle is on the first slide of every module. So what did we learn this chapter? So we learned that snapshots are a fast and convenient means of preventing regression. But just like never eating is a fast means of preventing weight gain, it's far from perfect. In snapshot tests, the vast amount of the work, the creating of the snapshots, the comparing of them, is done automatically or, dare I say, automagically because so much is happening behind the scenes. All the developer needs to specify is what to test and when. We learned that failing snapshots can be fixed instantly simply

by updating them. And we saw that snapshots, they don't guarantee that the component works correctly, they only prevent it from going back to a state that it was before or going forward to a state that you didn't intend. Coming up in the next module, we'll learn about testing React components. This will be the titular module of the course, and it's really going to be a great module. We'll learn about stateless and stateful React components and how the testing process differs. If you did watch the course, Isomorphic React, you'll know that the application is built entirely with stateless components, which I greatly prefer. So we'll have a look at how React Redux creates React applications that are basically the most testable. It's just really, really great. But then also, I know that sometimes you don't have the choice of using a stateless component, so we're also going to go step by step how to make a test for a stateful component. I even wrote up a stateful component just for you, the equivalent of walking on hot coals for me, but that's how committed I am. And we'll also have a little chat about Enzyme, which we're not using, but that doesn't mean it has no value. And you should definitely at least understand how it all fits in. So we'll be covering all that in the next module. We'll really be testing our components. And by the end of it, you should basically be good to go for any Jest projects. It's going to be great, so I'll see you there!

Testing Components

Introduction

Hello and welcome to the culminating module of this course, Testing React Components. I say culminating because it's going to take all the knowledge of everything we've learned so far in the course to really understand this topic. This is the most complex topic related to testing. So as I like to do, let's start with a question. That question is, What does it mean to test React components? As we've gone over many times, testing involves making sure that components do not regress. So the first thing that we check is that the output of the component hasn't unexpectedly become something else. So in this sense, we can look at testing React components fairly generally as something that we do to prevent regression. However, there's more. In addition, we use tests to deal with our rarely occurring corner cases. For example, let's say we have a component that shows how much money the user has in their bank account, but the money can show the currency in any amounts, and the user can have currencies in any amounts. So in real life, the odds are that a user who would have currencies in, let's say, equal amounts of Australian,

Japanese, and Belgian dollars wanting to view his whole balance in Bitcoin is probably pretty rare. It doesn't happen that often, but maybe there's a unique bug or a unique message that's supposed to display when that happens. So we can use tests to engineer these unusual scenarios and test our component's corner cases. Additionally, some React components generate side effects, meaning the components themselves make a call to an API or update the state of the application. If we could avoid generating side effects in components, then we would. But sometimes that's not up to you. So in this case, we have to somehow test that the components are making the right side effects or sending the right message out to the app or the outside world, but at the same time, we have to somehow not execute them because actually executing side effects can be time consuming and costly. As you can see, this is no trivial or simple matter. Hence, testing React components is not easy. Finally, there may be a variety of different interactions that a user can do with a component. So unlike a function in which you just pass a value and get a response, there may be several different buttons, drop-downs, and other related widgets that the user can interact with. Moreover, interacting with them in a different order or with even slightly different ways of interacting like, for example, swiping versus touching, can all lead to different things. So there's a wide variety of if-then situations in our components that we might want to test. So that's what we're getting ourselves into. Over the course of this module, we're going to deal with each of these points and explore some testing strategies that help us mitigate the complexity of them. First of all, though, let's discuss how we can make components which are testable. We'll do that in the next clip.

Building Testable Components

In this clip, we'll discuss constructing React components that are testable to begin with. Now it's true that you may not always have control over how the React components you're making and testing are constructed. But nonetheless simply constructing components in the right way is by far the most effective means to ensure successful testing. So one of the things about React that we're all familiar with is that there's a wide variety of ways that you can make a component in React. And when you make components in different ways, they're not necessarily the same. Some of them have different properties. Let's have a look. So components may or may not have lifecycle handlers. You're probably familiar with these such as componentDidMount or even Render, componentDidUnmount, etc. Stateful React components have these, and stateless ones don't. Additionally, components might have an internal state or not. Once again, the type of React components that we often considered stateful have this internal state, so they can have properties such as what element of a drop-down box is selected that are kind of kept private

from the rest of the application. As we discussed, components may or may not generate side effects. They may directly call APIs or other services from their componentDidMount method or a variety of other lifecycle handlers that are designed to do these things. All of these are, well, bad for testing. Each one of these things that you do makes it harder and harder to test your components. However, these programs are not without their advantages because it is significantly easier to quickly mock up an application using stateful components and all the lifecycle hooks they offer. It's just a lot harder to test them afterwards. Lastly, components might get their state from external arguments, which we often consider props. Unlike the first three, this is a good pattern since it makes it a lot easier to test our components. After all, we can specify what the props are wherever it is that we're defining the JSX that makes our component. Therefore, let's extrapolate on what we just discussed. There's a spectrum of React components, right? Well, which components on that spectrum are the most testable? So no internal state--whatever comes out of the component is idempotent or pure based on the props that go in. So if I have a certain component, and I give it the props 1, 2, 3, no matter what the other variables are in the situation, the output of that component will always be the same. If I want a more nuanced component, then I go with more props. Next, no side effects--it's much easier to test our side effects if they're handled by sagas or thunks or even services, anything but the components themselves. Now we will be building a component that actually does it and test this, but at best testing of side effects that come out of components is shaky. So this one particular pattern causes a lot of problems. Finally, on the same vein, no lifecycle hooks at all--lifecycle hooks like componentDidRender are arguably unnecessary. They do not exist in a React Redux-type structure application. We've all seen that these applications can be just as full featured as React applications using stateful components. So if we don't even use these lifecycle hooks, then we no longer have this problem or the previous problem of side effects. And our problem of internal state is also largely mitigated as well since we're not nearly as tempted to start fiddling around with an internal state when we don't even actually have any hooks that we're writing code in. But speaking of React Redux, is there a relationship between React Redux and Jest? It turns out that they go great together. So we will investigate that in the next clip.

React Redux and Jest

So let's discuss React Redux and Jest together, or rather than having an unbiased discussion, let's just praise the combination of the two because they are outstanding. Now, of course, this clip takes place in an alternate perfect world where the project you're working on is built with React Redux, or you're starting it from scratch tomorrow. If you're already stuck working with stateful

React components knowing just how much better things would be with React Redux, it may not immediately help you. But in an upcoming clip, we're going to learn about testing stateful components as well, which I put together exclusively for those individuals who do not have the chance to work with React Redux right now. But let's continue. So in React Redux, components don't generate side effects. The component connect container-type syntax really clearly limits us to dispatching actions. And actions on their own don't do anything. They don't generate side effects until they're caught by a thunk or a saga, a reducer, etc. So, in fact, if you try to generate a side effect within the code of your connected component, you'll find it's pretty hard to stumble into this anti-pattern. Additionally, each component consists of a logical pair of elements, a display component that we can easily test using snapshots and kind of test visually and container components, which are kind of this wonderful abstraction that let us do the thing in the previous clip where we want to test our side effects without actually doing them. So container components make this very easy. Additionally, React Redux components simply don't have internal state. Everything comes from the mapStateToProps provider. So once again, even if you are quite determined to execute this anti-pattern and make a component that was hard to test, you'd actually have a pretty hard time doing it with React Redux. It's just that good for testing. So what are the key takeaways, what are the tools in our toolbox that we're going to use to test React Redux components? So, first, we're going to take advantage of the fact that the container and display elements are naturally separated, and we'll test them separately using different techniques and different kinds of assertions. We're going to use unit tests to verify the container. We can run all the unit tests we want to make sure that the container does its job, which is basically successfully mapping the state to props and then mapping interactions to dispatching actions. And then for the display part of the component, we're only interested in the correct output of the React. So we can use the snapshot test that we've already learned to verify the output of the display component. Well, that was a lot to take in, but let's go ahead and start a demo where we're going to do this in the next clip.

React Redux Container Testing Demo

In this demo, we'll be testing our React Redux-based React component. So we'll add a snapshot test to the display element of the React Redux component. This will protect our component from regression, and it's a really easy test to write. Next, we'll add unit tests to the container element. This is quite a bit harder as unit tests require some critical thinking, whereas a snapshot test requires much less. However, it's still a lot easier than testing a stateful component. At the end of the day, we'll be able to see how working in this way the two different kinds of test provide

excellent coverage for our component. So here I am inside my text editor. In this clip, we're going to be writing some tests for the QuestionDetail component. So here in my application, you can see that if I click More Info for any question, we go to this question detail element, which is kind of just a---it kind of just displays the body of the question and lists the number of answers. It's not a very fancy component, although it does make up an important part of our application. So we need to make sure both that the container is mapping the application state to the props correctly and that the component itself is outputting the JSX HTML code that we expect. So let's go back to the editor. So in the components directory in the __tests__ folder, you can see I've already made the stub of QuestionDetail tests. Wow, we were learning about describe and it, so now we will write some tests. So let's start with the container. Now in order to test the container, we're really interested in mapStateToProps and mapDispatchToProps because, after all, these are the building blocks that make up our connected component. So if I go to QuestionDetail, we can see I have my display, which takes the form of a function, and mapStateToProps, which is also a function, and then we export this default connect. The first thing we're going to do is we're going to export our component and our mapStateToProps. So I'll export this. It's very cool that we can do this, and it's telling that to build our application, we didn't need to export these before, meaning nothing was and could have been depending on them. By virtue of the fact that we're exporting them for our tests, it kind of reveals how perfect this structure is for testing. So now that we've exported these, we're exporting this mapStateToProps method. We can now import it and run assertions against how it's supposed to be working. So here in QuestionDetail, I'll import mapStateToProps from QuestionDetail. That's looking good. And I will put another describe block in this. Of course, I'd like to say again that describe blocks are optional. I really like to use them to surround all my it blocks, but you don't have to, and I can easily imagine someone making the counterargument that they're just more code. So next we'll put a block where we describe the container element. And if you can believe it, yes, I'm actually going to nest another describe block in this just because I feel it mostly describes what I'm trying to do. We'll say describe('mapStateToProps'). This leaves the door open later for us to put a describe('mapDispatchToProps') if we ever add that. So now we write our tests, and this isn't a regression test. We're actually actively trying to figure out what's supposed to be output and then making sure that's actually output. So we'll say it("should map the state to props correctly"). So this is actually really interesting how we're going to go about testing mapStateToProps. MapStateToProps is a really simple map function. It just takes an object and spits out another object. So we'll define an empty object to be our appState and another object to be our own props or the props that were passed to the component. And then we can create a version of the componentState by calling mapStateToProps with these arguments. Fascinating! Let's see what

happens if we try to run these tests. I'll open my terminal up. I'll say jest questiondetail. Outstanding! So we're getting a runtime error. It's saying it can't call find of this property questions that doesn't exist. If we investigate our code, we can see that this cleverly written application looks in the state. questions, a centralized repository of questions, for the question matching the ID of the ID in its own props. Therefore, if we make a state object that has a list of questions, and we pass an ownProps in with an ID matching that question, then we'll expect the mapState to have the same question that was in the whole application state. That was a mouthful, but it might be a bit easier to demonstrate. So let's define this theoretic sample question, the one that we're imaging is the question loaded in the component we're testing. So I'll say sampleQuestion. And for this test to pass, this sampleQuestion necessarily needs to have the same format as the questions in our app. So we'll say question_id, and we'll say that's 42, and we'll just give it a body and just give it some random sentence. Now for the appState, we can give it a property questions, which will be an array consisting of just sampleQuestion. And in ownProps, we can say that the question_id is just the number 42. Now below here, let's add a console. log. So we'll log componentState and see what happens. Very cool. So as you can see, it's logging what we've expected. It's mapped the state to these props, which are correct. All we have to do now is assert that the mapped state is essentially equal to our sampleQuestion. So we'll expect(componentState). toEqual(sampleQuestion). And let's run our test again. And all tests pass. Isn't that something? This is a simple example, but it really is a great template for testing components. It also does a really good job I feel of illustrating the advantages of React Redux and any similar kind of Redux-based state container. I literally cannot emphasize how much harder this would be if all our state and our components on display were all muddled up together. In the next clip, we're going to resume right in the IDE as we add tests for the display element.

React Redux Display Testing Demo

So here we are just where we left off in our __tests__ file. So now we're going to test the display element of the component. So here under describe('The Container Element'), I'll describe('The display element'). And then right inside, we'll just say it('Should not regress'). I have it here, so I'll just copy and paste it. Now we're going to need to import some additional things. We're not going to need mapStateToProps at all for this test. But we will need the display itself, QuestionDetailDisplay, which we exported in the previous clip. Then just like our snapshot test, that's what we're going to do, we'll need react-test-renderer and react. So now that our imports are correct, let's write a snapshot test. So we'll generate the tree for our component. We'll say

const tree = renderer. create. And in this, we'll pass a JSX component. So we'll make an HTML-style bracket here, and then say QuestionDetailDisplay. Conveniently, the output of this display is idempotent to its props. So we don't have to worry about scaffolding any application around this or creating any other special scenario. We'll just pass in the props, and we're ready to run our assertions. So we'll just put in a bunch of details for an imaginary question that doesn't exist. Title, let's give it a silly title. And we'll also give it a body, answer_count, and tags. You can kind of fill these in however you want. They won't affect the rightness or wrongness of this example. So, now all we'll need to do is expect(tree. toJSON). toMatchSnapshot. Now let's try running our tests. So our tests pass, and as you can see, it wrote the snapshot. As we already know, the first time we run a snapshot test will never fail because that's when it writes the snapshot. We can easily verify that if we change the props that we're passing in here, so let's make the body of this 43 and change the title to THANKS!, now if we run the tests, well, they fail now. The snapshot test fails because the component's output is no longer as expected, and we can, of course, fix this by saying jest questiondetail -u to update our snapshot. So that's it! We've not girded our QuestionDetail component with a variety of tests. Unfortunately, life is neither easy nor simple. It is not like React Redux components. So we are not done yet until we talk about and demonstrate the testing of stateful components, which will begin in the next clip.

React Renderer vs. Enzyme

So before we proceed, I thought I should clear something up--Enzyme versus React test renderer. I have several times mentioned my preference for React test renderer. But if you're a React developer who's been working anywhere in the field for the past 10 years, you might have expected me to automatically use Enzyme. Well, why'd I make this decision? As Napoleon Hill, author of "Think and Grow Rich, " once penned, "Neglecting to broaden their view has kept some people doing one thing all their lives. " Well let's broaden our view. React test renderer versus Enzyme--what's the comparison? So React test renderer takes a React component and outputs the resulting HTML, but it doesn't need a DOM to do so. It's from the same group of people that developed React themselves. So no one has a better understanding of React than these individuals. It's useful as we've seen for getting the output HTML of a component when we're snapshot testing, and not a whole lot else. Finally, if we look in the Jest documentation, this is the renderer that they recommend. Now let's compare that to Enzyme. Just like React test renderer, Enzyme takes a React component and outputs the resulting HTML. It doesn't need a DOM to do so. So these services are the exact same. Enzyme isn't from the React team, but it's from the Airbnb team. Industry-wide, people have a fair amount of respect for the Airbnb team. They may

not have quite the monopoly on front-end libraries as Facebook or Twitter do, but they're still a team that's done some good work. So Enzyme also has a bunch of other additional features that are useful for testing components, especially stateful components. So, for example, Enzyme has an API to manage clicking or to simulate keyboard input. So if you had a stateful component where if you did the Konami code, up-up-down-down-left-right-left-right-a-b-a-b, and then something special happened and you want to test that, you could use Enzyme to simulate those inputs. That's kind of beyond what React test renderer does, and you'd need to have kind of a smarter system of side effects and services for testing that in a connected component. Unfortunately, sometimes these features that I've mentioned simply do not work, and that makes using Enzyme a real challenge. So as you can see, Enzyme--here we are the GitHub repository for Enzyme--Enzyme has almost 300 open issues, which are generally improvements or sometimes bugs that people point out with the application. If we click this, so if we have a look at the labels of this, we can see that of all these issues, 30 are open bugs. Now the problem is 30 open bugs is just too many open bugs to have in the library that you're trying to use to make sure that your code works right. When your application isn't working, and you try to fix it by writing tests, but then you uncover a bug in your test suite, it is incredibly frustrating. So based on this, I've gone with the somewhat uncontroversial decision of going with the tool recommended by the Jest team, which is React test renderer. But if you're ever in a discussion with someone and they ask the obvious question, Hey, why aren't we using Enzyme?, this is one of the answers that you can give them, There are too many open bugs in the Enzyme repository. Some of these have been hanging out here since early 2017. So what can I say? Don't use Enzyme, at least not until they fix some of these issues. I personally have spent hours, perhaps days, trying to fix a bug in my React test only to realize that it was a bug in Enzyme. I don't want you to waste the same time. In the next clip, we'll be diving in to testing stateful React components.

Testing React Components

So testing stateful React components. This is the ultimate challenge for any student of testing. Not only is it challenging and rife with trickiness, but you'll need to combine everything that you've already learned to understand what's at stake here. So, first, you'll notice that during our React Redux test, we actually didn't even have to use Jest's mocking API at all. Mocks just aren't necessary with React Redux components. However, we do need to use mocks for our stateful React components. Our react components are going to be requiring all kinds of services and making all kinds of calls to them. We don't really have any control over this behavior. We just have to mock any and all services that we need to mock for our component to react a testable state--

neither clean nor easy. We'll be using spies to verify our side effects, so rather than verifying that, say, the dispatched action is correct, we'll actually be mocking, say, our fetch user API, replacing the method with a spy, and then checking on that spy to see how the method's been called. In this way, we'll assert that our side effects are happening without them actually happening. Now the problem of having logic in our lifecycle hooks is so severe and makes things so hard to test that we're actually going to move the logic wherever we can from the lifecycle hooks to services. And a service is basically just a JavaScript class which provides various forms, usually data or sometimes data processing idempotent functions, etc. Just like with stateless components, we'll use snapshots to prevent regression. Now to actually get our React component to look the way we want, it's not simple like React Redux. We need to kind of scaffold an imaginary app around our component. We have to do as much and anything that it takes to convince our component it's where we want it to be. And, finally, as they say when you find yourself at the bottom of a big hole, stop digging. The best way to test React components is as soon as you can, you might not be able to go back in time and change the things that you've done, but moving forward whenever you can, make stateless components. Alright, in the next clip, we will do the ultimate demo of this module. It promises to be an outstanding couple of clips. I'll see you there.

Building a Stateful React Component

So this demo will consist of two parts. First, we're going to create a simple but stateful React component. This basically involves me from the beginning making a React component in the manner I would consider to be wrong since for the original isomorphic React application, I made all my components in a way I would consider to be right. That's great, but we can't demonstrate how to test stateful components on a correct stateless component. So we're going to sit down and actually make a stateful React component. So that'll be the first part. Then in a second part, we'll write the test. So we'll create a manual mock for a dependency of the component. We'll write assertions regarding that component's output. And we'll set up a few snapshot tests to prevent regression. So here I am in my IDE, and I'm going to make a new component here in the components directory, and I'll call it NotificationsViewer. jsx. So this is going to be a stateful component. Basically, it's just going to output a number, and that's the number of notifications that a logged-in user has. We're going to be mocking up this information not using a real API as you'll see shortly. So we'll import React from 'react'. So to create a stateful component, we cannot use the anonymous function syntax, which I so prefer. So we're going to use the extend syntax as follows. We'll say export default class, and we'll say this class_extends React. Component. So now we have a React component. So we'll give it a constructor, and the constructor will take an array

of arguments. And we'll just call super with that array of arguments. And we'll define the application state. So we'll say this. state. And the state just consists of a property count, which is how many notifications you have. So it'll just be count. And I'll make the default number a -1. Now we'll add a render method. So render returns JSX code, and whatever that return code is evaluated as appears on the screen. So we're going to write this, and it'll just display how many notifications the user has. So we'll just add a section, give it a few bootstrap classes--a margin top 3, margin bottom 2. That looks good. Then we'll have a div, and we'll give it a notifications className just in case we want to use that className for testing later. And we'll say this. state. count, if it's not equal to -1, show the count. But if it is equal to -1, we can't have -1 notifications. That doesn't make any sense. So we'll just say Loading. So let's see how that's looking. Let's go to App. jsx. Here at the top, we'll import our NotificationsViewer. And here in AppDisplay between the top div and the top route, we'll just have the NotificationsViewer in its own little div. We don't need to pass any props in right now, so that's just like that. So here in the application, you can see our new component has made a little appearance here. It's this Loading word. Right now, it doesn't really do anything because it's not complete, but there it is. Let's complete it now. Back into the IDE. So here I am back in NotificationsViewer. Lastly, I'm going to use a lifecycle hook to fetch the data. So we're going to use a service, we're going to simulate a service type structure where during componentDidMount, it's going to call this service, and the service will return the number of notifications. So we'll import NotificationsService from, and we'll just say '. /services/NotificationsService'. And you're probably thinking to yourself, Hey, Daniel, this file doesn't exist yet. You're right, it doesn't. Let's create it now. I'll copy that, and I'll make a new file in src, services/NotificationsService. js. So here in NotificationsService, I'll export default. So it's an object, and the object will have an async function called getNotifications. Now, first, I'm going to put a console message in here, and the message is going to warn us really clearly that this is the REAL NOTIFICATION SERVICE! It's really contacting the APIs. And then we will await delay of let's say 1000, so we're simulating a delay, and let's import delay from redux-sage. And then, finally, we'll just return this hardcoded value. So we'll say count is, I don't know, 42. So as you can see, the way this app is written now, there's no way for us to avoid calling getNotifications, getting this REAL NOTIFICATION SERVICE message. How are we going to fix that? Don't worry, there's a way. So back in the NotificationsViewer, I'll just change this so it's one directory down. Great! So I'll define an async componentDidMount method. So we'll say let (count), and then we'll await_NotificationsServices. GetNotifications. Then we'll call this. setState just passing in the count. And I made one little mistake here. This GetNotifications has a capital letter. This doesn't. Let's make them consistent. I've been writing a bit too much C# lately, folks. Very cool. So if I go to my actual application, and I'll give it a refresh, you can see first loading appears for one second,

and then the number 42 appears. So this is in effect our stateless component. You know, there was one thing I want to fix; 42 on its own doesn't make sense, so let's go back to this component. And let's just change this to state. count Notifications Awaiting! So now it's at least a sentence. Let's go back to the app. Great! So it loads, then the service returns this, and it says that. Now if you're wondering about that flash of unstyled content, that's from the isomorphic side of the application. That's the server rendering showing how it does not update its code as quickly as the client rendered version. And there we go. So we've spent the last 10 minutes or so making a component that I would consider hard to test and debatably wrong. So in the next clip, we're going to test this component. It's going to be a toughie, so stick with us.

Testing a Stateful React Component Demo

Well, folks, here we are. We are doing the final and I'd consider the most difficult demo clip of this course. This is not the last clip by any means, but it will challenge in the most serious way your comprehension of Jest. What we're going to be doing here is we have this NotificationsViewer we just made. So we're going to write a suite of tests for it, and this is a stateful component. It is not easily tested. So we're going to need to use a lot of tools in our special box. Let's begin with the obvious. I'll make a file in components/__tests__ called NotificationsViewer. As always, you could make this NotificationsViewer. spec in the same folder as our JSX file. There's really no right or wrong answer there. So we'll describe("The notification viewer"). So we're interested that it displays the correct number of notifications. If the user has 5 notifications, it should say 5 notifications. Not rocket science, but how easy is it going to be for us to actually test this? We'll say it("should display the correct number of notifications"). And we'll define this as an async function. So let's do a few imports. We'll import the viewer itself, just the file we just wrote. Nothing too fancy there. And then we'll also get React test renderer, React, and our favorite utility delay. I'll just type those in right now. So let's just try to naively write this test. We'll say const tree = renderer. create and pass it in an instance of the NotificationsViewer. We will await a delay, which is kind of just a hack that we need to do to let all these lifecycle handlers finish before we test them. And now what we'll do is we'll use the following code to isolate a particular part of the component and test its output HTML against what we expect. So this is kind of like a snapshot test, but we're really much more saying, I thought about this. This is what this should equal. This is a thoughtful test. So we'll make an instance or, rather, we'll describe the instance as being the root of the tree. So we'll say const instance = tree. root. And we can run methods on this instance. So we can say const component, and then we'll call the findByProps property of our instance. So here we could use any props to find a component. As you'll recall, we put a className on a

component containing the notifications for convenience, and here we go. So we'll find it with the prop className: 'notifications'. Then we'll get the text, which is the first element of the component. children array. And we'll expect the text to equal---and we don't really know what's going to happen next. We're just kind of going for it. So let's just put some answer that we're hopeful is close--5 Notifications. So now let's see what happens. I'll say jest notifications. Wonderful! So we can see that our naïve test has unearthed a large number of errors. First of all, it doesn't say 5 Notifications. It still says Loading. It never quite gets past that point in the short amount of time we give it because the real NotificationsService needs longer. Additionally, we're seeing this console warning, REAL NOTIFICATION SERVICE! REALLY CONTACTING APIS. This should alarm us because that means that every time we're running these tests, we're really calling these APIs. That could be an expensive mistake. So can you guess how we can solve all these problems? That is correct. We can mock the NotificationsService. It's a lot of work, but it's our only option at this point. So let's go to services, and we'll create a new file, and we'll call it __mocks__/NotificationsService. js. So we're now going to define this class, which is our mocked version of the NotificationsService, an evil clone with no internal workings that is meant to convince other modules that it is our real service. So we'll define a variable here at the window level called count, and this is where we'll be storing whatever value this evil clone has been instructed to parrot out to whatever asks. So I'll export a default. We'll give the same message, async GetNotifications, and we'll also give it another method called __setCount. And you pass it in a count, and it just makes the global count that count. And the dunder indicates that this is a private or almost semi-hidden method and also implies that this is not a real method of the real NotificationsService. Then for GetNotifications, we'll put a different console. warn("GOOD JOB! USING MOCK SERVICE! "). And we'll just return count. So that's it. There's our first mocked service. Pretty great, huh? Let's run our test and see if anything has changed. As you can see, we get the same message, REAL NOTIFICATION SERVICE. But we wrote our mock, and we remember that when we mocked isomorphic-fetch, we didn't have to specifically describe the mock. Well, for local modules, you do. It's a tricky gotcha, but luckily with the code whisperer here to walk you through this process, it's fun and easy. So here in the NotificationsService right under my import statements, I'll say jest. mock, and all I'll do is pass it in the URL of the NotificationsService. It's at least smart enough to know that if I said this to look in the __mocks__ directory for this mock. I actually don't have to tell it where it is, I just have to say, Listen, for this test, I would like you to please mock this module. You know what to do. Now let's try running our code and see if anything has changed. Outstanding! So we're getting a different but expected error. Now insteaed of saying 5 Notifications as we expect, it's saying 0 Notifications plus the word Awaiting, which I kind of didn't remember I put in there, but we're going to fix that in a

moment. And we're also saying, GOOD JOB! USING MOCK SERVICE! So now we've at least gotten rid of the problem where our tests are really creating side effects. Now we're using the mock service. However, we want 0 Notifications to say 5 Notifications. How can we do that? So you'll recall that our NotificationsService had a __setCount method for just this purpose. So we'll use the beforeAll hook that we have. So right at the beginning just once, we'll say NotificationsService, and actually we'll have to require that. So we'll say const notificationsService = require('.. /.. /services/NotificationsService'). default. Now you may have correctly noticed that I've used import as every other statement but require here. Long story short, the behavior of import and require is not exactly the same. They especially behave differently when you try to put import statements after certain things or within blocks. They tend to get hoisted more. However, it's critically important that I require my NotificationsService after I've called jest. mock. So rather than saying jest. mock X and then import X, I'm saying jest. mock and then require so that the require statement does not get hoisted above our mock. So now that we have this, we can say notificationsService. __setCount, and we'll make that to 5. And I'll just update this here, so I add the word awaiting. Now let's run our code and see what happens. And everything's looking good. Let's just add a capital letter to the word Awaiting. And the test passes with our mock service message. We can easily verify the flexibility of this by changing the number that we're passing to the NotificationsService. This time, we'll try sending 42. When we run the test, we get an error saying we were expecting 5, but we got 42. Let's change this to 42 so the test passes again. Give yourself a big old pat on the back, take a deep breath out, and we're done. Testing that stateful React component was no walk through the cake factory. But naturally we just learned so, so much about testing React components right there. In the next clip, we'll summarize what we've learned.

Summary

Well, that was a challenge. Everyone take 5 minutes to refresh your coffee cups, and then once we're good and relaxed, let's summarize what we just learned. So we learned for the reasons that I recommended that React test renderer is a good call to use over Enzyme most of the time. We learned that testing stateless React components is really easy. It required no special setup, no mocking. In fact, we barely had to do any work at all. This is because the container and the display while combined expressing a whole component can themselves be tested separately. We learned that testing stateful React components is really hard and that in addition to the risks and challenges that go along with it just being stateful, we have to go about doing things like mocking external APIs. In fact, we have to devise whole evil clones for our services that allow us to coax

our stateful React components into whatever state we need them to be like we saw with our NotificationsViewer. But ultimately we saw that by combining everything we've learned in this course and a little bit of hard work, we can succeed in testing both of these component types. So what's coming up in the next module? The next module is brief. It's sort of a bonus module on Jest matchers. So we'll be asking some questions, What are matchers? We're going to kind of use it as an excuse to write a few more test files to test things out. And we'll get an idea of some of the additional expectations that are available in Jest. You know, it's nowhere near as challenging as what we dealt with in this chapter, but it's a good way to wrap up, and the demos that we'll be doing will give you a chance to cement what you've learned. So I'll see you in the next module.

Advanced Jest Matchers

What Is a Matcher?

Hello and welcome to this module entitled Advanced Jest Matchers. This is a brief module, a bonus module of sorts, in which we discuss the many matchers that are available in Jest, as well as write a few more tests in the demo portion. So what is a matcher? We've already used them quite a few times as we've been writing tests, but let's demystify it altogether. So you might also hear it referred to as an assertion or an expectation. So a matcher is like a way of saying in your code that you expect something to be something else, and this goes beyond just expecting, say, the string A to equal the string A. You can assert that an array has a certain property or assert that something belongs to a particular class of things like falsey things. So it's not just necessarily a one to one relationship. And if you give a matcher a claim that it can't validate, it will effectively throw an error, which is what let's Jest know that the test failed. So to help understand what assertions are even more, let's look at this simple code example. So here's a simple function called test. It gets the value from some method called getValue42. And then it checks to see if that value is equal to what it expects, 42. If it's not, it just throws a new error. Now here's a test. Notice something? That's right. These two blocks of code are actually equivalent. On the top is a handwritten test in which we have an if statement which checks something and a block which throws an error. However, when running our application, an error causes the JavaScript code and effectively the application to halt. But with the test runner, an error just tells it that that particular test failed, which does not mean that the other tests do not run. Using a matcher, we can see the equivalent code below. As you can see, the test below is a lot clearer and easier to read. In

addition, it's clear that it is a test because it uses the expect family of matchers. The above function, it's hard to tell if this is supposed to be testing something or if it's actually a part of the system's functionality or even what throw new Error means in this context. Are we expecting the application to stop? Or are we expecting this to be dealt with like a test? It's up for interpretation. So in the next clip, we'll be exploring a few of the existing families of matchers that we have available.

Exploring Matchers

In this clip, we'll be exploring matchers. Jest has many matchers for us to use, and we'll be increasing the breadth and depth of our knowledge on this subject so we can use them more effectively. So here I am in the documentation page for expect on the Jest documentation. This lists all the matchers that are available. If we scroll down, we can see that there're really a lot of them. That's a pretty long list. Now believe me if I could, I would love to spend time with you discussing and demonstrating each and every last one of these matchers. A four-hour-long course, forget about it. How about a four-hour-long module? That sounds like lots of fun to me, but it might not sound like so much fun for you. So this is a page that you're going to want to take some time to read on your own. I've just picked a few of the best most relevant matchers that I wanted to talk about. If you would like, for example, to see a whole course just on expectations and matchers, which, by the way, goes beyond a simple React or Jest application, please let me know in the discussion or on Twitter. But for now, let us discuss a few of the matchers that I thought were the most interesting. As Shakespeare once said, "To be or not to be. That is the question. " In this case, that literally is the question. Do you want to use the to be expectation? Do you want to combine that with the not expectation? Let's find out because, after all, we do have great expectations. So not. The not matcher is a very special matcher. It reverses an assertion. So it works with any assertion and kind of inverts it and turns it into its opposite. So, for example, if you have an assertion that says, for example, assert X to equal 5, it will match on number 5. If you change it to assert X not to equal 5, it will now match on everything but 5. So it reverses it. Personally, I don't like to use the not matcher in my tests because it's kind of hard to understand. I mean, listen to the sentence, I won't not remember to not go to the store and not forget not to not buy milk. Did you clearly understand what I was saying? Was it clear in my sentence if I was going to get milk or not? I don't blame you if you don't know the answer. I don't even know the answer either. My point is that by adding the word not to an expression, you essentially make it harder to understand. So, I'm not a big fan of not. But you may disagree. You might think that it's great, and it's there for you to use. Next, to be and to equal. These are kind of the quintessential

matchers, but they differ in kind of weird ways. So to be and to equal verify that two values are equivalent. But they don't mean the same thing, and a lot of times to be will be false, but to equal will be true. For example, let's take two different arrays with the elements 1, 2, and 3. As far as Jest is concerned, these two arrays are equal, but the first array is not the other. So to equal would pass, but to be would fail. And in general, you want to use both of these operators as rarely as possible because they're so general. For example, you can easily compose any of the higher level matchers out of the to equal matcher. But the point of those matchers is to make your tests more readable. Therefore, the more you use to equal, the less readable your test is unless in each and every case, there really was no more appropriate matcher. So there's one last family of matchers, but to understand them, you have to understand this quote from Douglas Crockford, who said, "Unlike most other programming languages, there is no separate integer type (in JavaScript), so 1 and 1. 0 are the same value. " So that brings us to the to be close to operator. So to be close to is like to equal, but if the numbers are not quite equal but very, very close, it still passes. This is useful for any assertions involving floating point numbers. Here I am back in my browser, and just pay attention on the right here to the console. So here I have my regular JavaScript console, which you can open up in your own browser if you wish. So let's look at some numbers. I'm going to type the number 3. It makes sense, right? Now let's type in a number that's definitely not 3. So let's say just 2. 99999 repeating a lot of times. So as we can see, 2. 99999 is 3 as far as JavaScript is concerned. So we can say (3 == 2. 99999999), and just look at that really closely and ask yourself, does 3 really equal 2 with about 20 9s after it? And, of course, it doesn't, but JavaScript says that is true. So JavaScript can't tell that this 3 on the left, it looks like an integer, and the one on the right looks like a float, and they're obviously not the same. But you asked JavaScript, and they are the same. JavaScript has loads of such intricacies. And that's why matchers like to be close to exist. Finally, let's talk about to contain and to have length, which we're going to use in the demo clip. So these are array matchers, which are very useful because we're often trying to assert stuff about arrays. These matchers verify things like the contents of a collection, as well as the size of it and if it matches another collection. As I mentioned before, we can accomplish all this with some arrangement of to equal matchers, but these provide a much more succinct way of doing that. Basically what I'm trying to say is using to contain or to have length doesn't offer any strategic advantage or new features. We're using these just to make our code more readable, more readable for ourselves and for other developers. Just like normal code, after all, tests need to be maintained as well. So that's a brief walk through the world of matchers. In the next clip, the final demo of this course, we will add some additional tests and test these matchers out.

Jest Matchers Demo

So welcome to the final demo in which we'll really be jumping into Jest. We'll be adding some unit tests to our reducer. This isn't technically a React component, but nowadays, you're pretty likely to find reducers in any given React application. So this isn't far off the mark. We'll be using the toContain and toHaveLength matchers for our reducer since our reducer works with arrays. So this is a great opportunity to try them out. In addition, we'll use the not matcher to see what happens when we reverse some of our assertions. So here I am inside my text editor. We're going to keep things really simple in this clip, so we're just going to be writing tests for our reducer. Reducers are very easy to write tests for as they are pure functions, which are designed structurally to be testable. They're great. So I'll make a new file in my reducers folder, and I'll call it questions. spec. js. Let's put describe and it blocks. So we'll say ("The questions reducer") it("should work"). And, by the way, in production, you may not want to name your tests "should work. " This is a common name, but it doesn't really give any information about the test. So once we've written our tests, we're actually going to try and rename this so it's more descriptive. It's alright to have this dummy name as we're writing the test. So in questions. js, you can see we're conveniently exporting this questions reducer, which, again, is a pure function. It has no dependencies. So we can test this really easily. We're going to import this into our test file. Now let's just add a log. And to make sure this is working, let's run jest questions. We see our log. We don't see any errors. We're ready to proceed. So, first, let us test that if we give it an action that it's not equipped to handle, it just returns the original state. So we'll define the state as an array. And I'll give it two values. You can give it any two string values you want. You're allowed to have a lot more fun that I am. I'll just choose some obvious values, foo and bar, and we'll get the new state by invoking this reducer with the original state, and then we'll just pass it an action type that is essentially nonsense. So now we have this new state. So let us run an assertion. We'll expect the new state to equal the old state. So we'll expect that to equal array. I made a little mistake here. Calling this array sometimes and state sometimes. Let's call this state everywhere. Good, let's run our code again. So no errors so far. That's looking good. So the new state is indeed equal. What about the toBe operator. Is that going to work as well? It is. As expected, if we look at our reducer, we can see really clearly that here at the bottom if it doesn't match any of these, it just returns the same state. So it's returning the very same object. Here, let's do something interesting. Let's define this state clone. So it's a different array with all the same properties as state. The first item's foo, the second item's bar. It has two items. Now let's see how the toEqual and toBe operators like this different value. So we'll say expect(newState). toEqual(stateClone), toEqual. And what happens? Just rearranged my windows a little bit, and you can see that it does indeed

pass. However, let's try (newState). toBe(stateClone). So we get an error, and it says---it's actually a very great error. And it says that these two values have no visual difference. But yet they don't match. So we actually get this nice block of text from Jest explaining exactly what I'm explaining now, that toBe and toEqual are not the same. So here we can leave this as expect(newState). toEqual(stateClone), but expect(newState). toBe(state). And now that we know what this test contains, we can rename this test block. It should not modify the state for an unrecognized action. So now let's test our reducers functionality when we're adding new items. So this time, this test is going to have a valid applicable action type. So we'll say it("should add new questions"). So we'll define an existing state, which will be an array of objects. Each of these objects need to have a question_id property, and I'll leave those values up to you. Now let's define a newQuestion which has the same format but a different value. Looking good. Now we'll get our newState by calling the reducer, which are our existing state and the correct action. So the action type will be FETCHED_QUESTION just like in the real app. And, yes, I should point out that many applications like to keep constants like this in a central file and get them with the import syntax. I find that because of the way JavaScript works, these import statements don't work correctly 100% of the time, so usually I omit them. But it's a very common convention that I'm aware of, and it's fine if that's the way that you're doing it. So we'll give this action a question property, and that question will be our newQuestion. And for now, we'll just console. log our newState and make it a little bit clearer what we're asserting. I'll open my terminal up. Great! So we get no errors, and we can see that our new state kind of as expected is foo, bar, and baz, with baz at the beginning. Kind of weird how the questions get inserted at the beginning, but that's just how the code is written. So now let's run some assertions. First, we'll say expect(newState). toContain(newQuestion). Pretty easy, right? Let's run our test. We can confirm that passes. Now let's mix that up, and we'll expect the existing state not to contain the new question, which is kind of a valid test because we're kind of verifying that the reducer is not modifying our existing state. Let's run our tests again. Outstanding! So there's not. While we're at it, let's verify the length of our new array. Great! So all these assertions passed. We can get rid of this statement. Now let's just test one thing. We'll make a newQuestionClone, which is the same structure but a technically different object. So it's equal to newQuestion, but it is not newQuestion. Let's see if the toContain matcher works like toBe or toEqual. Great! So we can see that toContain does not match correctly on this value. It actually works just like toBe. Fascinating! So we'll change this back. And there we have it--toContain, not, and toHaveLength. So, once again, I really enjoy doing demos like this, but this is simply all the time we have to spend on matchers. And this also concludes this module and essentially this course. In the next module, we'll be just concluding wrapping up.

We've got some great tips on what you can learn next. It's really cool, so I'll see you there in the next module. Thank you for watching.

Conclusion

Executive Summary

Hello and welcome to the concluding module of this course. It's been a great course and just packed with information. So let's begin this conclusion with an executive summary, sort of a summary of everything we've learned. So if you're going to take away anything, what are the key points that you should take away from this course? So we learned that testing adds work. Writing tests is a big of work as is maintaining them. But we also learned that tests save time and perhaps more importantly money by preventing regression and preventing the excess work that goes along with that. We learned that as far as test runners go, Jest is just an all-around great choice, not just for a React app but basically anything JavaScript based. And if we're trying to decide what test runner to use for our React app, the decision is even easier since the React team recommends that we use Jest. So we learned about mocks, and we learned that a mock has the same methods as another object. It looks just like that object, but it has no side effects. And we saw how we use mocks and their unique properties to facilitate testing. In the module on snapshot testing, we learned that snapshot tests prevent regression by warning us when components break or otherwise change in any way while we're working on the app. Of course, we also learned that developers have to respect snapshot tests in order for them to work. So they're a mixed bag. Finally, in the chapter on testing React components, we saw that everything about stateless React components, for example, the ones you'd make in a react-redux application, are much easier to test than stateful ones. But we also saw that if you have to test a stateful component, then Jest's many features, especially mocking, will come in handy. In the next clip, we'll discuss some ways that you can continue to learn after completing this course.

Continue Learning

So you've reached the end of this course, but you don't want the learning to stop. A commendable sentiment. Don't worry because we've got you covered. Here are some great courses that you can start watching as soon as this course is over. So, first, I'd like to recommend my recent course, Redux Saga, on Pluralsight. As you can see, the course, Redux Saga, is available

here at the URL you can see at the top of the screen or just search it on Pluralsight. In my course, Redux Saga, we basically cover everything that you need to know to integrate Redux Saga into your existing Redux applications. And then we also use it to do some really cool things like manage our external ajax calls and any sequential things that need to happen inside the application using sagas. So we had one or two sagas in this course that were there in the demo project, so if you thought those were cool and want to learn more, taking my course, Redux Saga, is probably the best way today for you to learn about sagas. So go ahead if you're watching this course, that means you have access to the course on Redux Saga. So I really recommend it. So, got thoughts you'd like to share about this course, about Jest, about React, about anything technology related at all? First, remember to share your feedback and rate this course. Next, I recommend that you star this repository on GitHub, the course's repository, and share your contributions by opening issues and pull requests. So the repository is called isomorphic-react as you can see, and if you want to star it, just click here. This lets other people know you think it's an important library. Finally, don't forget to follow @danieljackstern, that's me, on Twitter and share any news about relevant technologies like React or Jest that you might have. So with all these ways to join the conversation, you can always feel that your voice is heard. In the next clip, we will wrap up the whole course.

Thank You

Wow! It has just been such an outstanding course, hasn't it? We've all learned so much about how to be better programmers, but there's more to it than that. Working through this course was probably hard. You probably spent as many as eight hours of your own free time working through this course and completing the examples. I appreciate that. You could've spent that time doing anything, but you spent it watching my course. So for that I thank you. Moreover, you've demonstrated to me that you have the determination and are serious enough about web development to really go the extra mile and succeed. So from me and everyone here, I'd like to say, Thank you, and we'll see you in the next course.

Course author

Daniel Stern

Daniel Stern is a coder, web developer and programming enthusiast from Toronto, Ontario, where he works as a freelance developer and designer. Daniel has been working with web technologies since...

Course info

Level	Intermediate
Rating	★★★½★ (147)
My rating	★★★★★
Duration	3h 36m
Released	11 May 2018

Share course