PART 1 – Intro to React Testing with Jest   
Testing is an important part of any applications development. Understanding the testing tools for react is therefore an important part of the skillset. We'll look at the testing tools that are built into a project set up using create-react-app. Including the Facebook Jest testing library, and some other packages that can be used to help us test. We want to be able to unitise components, gaining an understanding of what to test and how we can test it. Including snapshots, dumb testing, and event testing. To do this, we'll examine how our Jest test file is structured and the building blocks we're using including suites specs, set up and tire down. To be able to make appropriate assertion in tests we'll look at the in-built matches. For testing functionality, we'll examine mocking and spying on functions and for unitisation we'll visit mocking components. Test reporting is also known important aspect. So we'll demonstrate how code coverage reports can be generated and viewed. As we're focusing on testing react, we won't be looking into any integration or end to end testing as this really doesn't concern the react code we write. We'll also points out what you should and shouldn't test and why.

# PART 2 – How to set up the Test Environment

 If you've started a project using create-react-app. You've already got everything you need to start testing your application setup. It comes with jest baked into the setup and a sample test spec for the app component. If you examine the package.json file, the command test is already in the script section on line 15. And you can see that testing library jest-dom, react and user-event have been added as dependencies, on line six to eight. This is the package.json file from a new app scaffolded with create-react-app. The command npm test can be used on the console to run the tests. And these will hot load as you make changes to them or the files they test. We press A to run all of the tests, and you can see that the default test provided for the app component passes. Tests should have .test. in their file name as a convention, to help identify them. But create-react-app is to configured, to look for tests in any file and to use them when running tests. For this reason, the test files don't need to be saved in the same folder as the item they were testing. In fact it could get mighty confusing, if you don't follow the convention. We've moved the app.test.js file into a folder we've called \_\_testing \_\_. Those underscores help to visually identify it as non-production code folder. And there's another convention. If you haven't used create-react-app. The react and jest documentation explain how to add testing to your project.

# PART 3 – Jest

Facebook claim that Jest is a delightful JavaScript testing framework, with a focus on simplicity. It works with projects using Babel, TypeScript, Node, React, Angular, Vue, and more. It's Open Source, and its capable of testing any kind of JavaScript application. It can follow Behaviour-Driven Development to ensure that each line in the JavaScript is properly tested. Its syntax is small and simple, and it can test the smallest part of an application as a unit. The main advantages of Jest are that, It doesn't depend on any other JavaScript framework or a library. It doesn't need a DOM to test, and it has clean an obvious syntax. As with many testing frameworks, it splits testing up into smaller chunks. A suite is used to define collections of similar type test cases written for a specific file and, or function. You can nest Suites to split files, and functions down further. Jest has files usually containing describe function. This is how a Suite is defined. Suites commonly have at least one test inside them. However, test files need not contain any Suites. The describe function has two arguments. The first is a string used to identify the test suite, and a callback function that contains the test specs. Tests are defined using either the it or a test function. there's no difference between them. If you check out the documentation, you'll find that it, is just an alias for test. They take two arguments similar to the describe function, a string to identify the test by, and a callback function that contains the steps to perform the test. The callback should have, some form of assertion at the end of it. Usually a call to expect. The expect function defines the actual value obtained by running the test. You can have more than one expect in a test. Although this is generally discouraged as a spec should only test one feature functional value. The second part of the expect call is to chain a Matcher function, to help compare the expected result with the actual result. Jest provides a large number of common built-in matchers like two equal. Each matcher, does a billion comparison on the actual and expected outputs. Here's a list and a set of descriptions for some of the built-in matchers. The full list can be found in the Jest API, and there's far too many for us to go into the more all we'll explain the ones we use as we use them. If you've looked through the built-in matchers, and none of them are appropriate, you can create a custom matcher as long as you follow the correct pattern. Again, the Jest documentation explains how you can create these, the testing functionality of Project Setup using create react app have a code coverage tool included. The switch minus minus, minus minus coverage is used on the end of the NPM test command. This generates a test report on the console, but also stores the results in a /coverage folder. This has reports in many formats. If you want the interactive HTML report, this can be found in the index.html file in the lcove-report folder. So there's an overview of Jest and how it can be used to test JavaScript. Coming up, we'll see how we use it specifically to test react components. So here's an example of a test spec that has one suite and three tests. The suite defines two variables A and B that are initially undefined. The first test defines A and asserts that it is true. The second sets A to false and defines B as a string, and asserts that the string mi is somewhere in it. The third test asserts A is true and not a falsy . when we run these tests, the first two pass and the third fails. This is because the values for variables used in test follow normal JavaScript scope. If apparent level value is changed inside a spec, it will change false following specs.

# PART 4 – The What and How of testing in React

React is for taking data and displaying it. So we should test any part of our application that takes data and renders it. There are two ways we can do this, render a component tree in a test environment and assert on its output or run end to end tests, which as we've already mentioned is not really concerned with React components, but what should we test? According to test carried Kent C. Dodds, "The more your tests resemble the way your software is used, the more confidence they can give you". A good rule of thumb is that you should test anything that does not duplicate the exact application code in the test, it's not a responsibility of, or covered by other tests. Meaning other libraries or reacts to core implementation, or if it has detail is important to outsiders. Meaning can they affect different API internal detail be described using only the components public API. With this in mind, we should be trying to write tests that check the output of our components rather than how they get to the output. The main reason for that is if you refactor your component and you test its implementation, the test will almost certainly break, even if the component produces the required outcome. There are a few strategies that I use to test React apps and some, or all of them can be used. The first is for components that don't often change in your application. The best way to test a component like this, is to make a representation of the DOM it creates and then check that it repeats this representation on testing. This is called snapshot testing, and we'll look into that first when we go more in depth into testing soon. The next strategy, is to render an initial component, perform some interaction with it, and then check to see if that we render results in the output we expected. There are a few ways in which that can be done, but we'll check out Facebook's recommendation for this. To preserve the unit nature of components, we need to understand how we can test those that have dependencies on others, and those that may render other components. To be able to do this effectively, we need to know about mocking strategies for other components and functions that we may import. We've introduced Hooks into our applications, and some of them are built in. There's no need to test the actual Hook, but if the Hook effects our component, we should check if behaves as expected. The flip to that is the custom Hooks we may have made. Like when we covered state management with a context and reduces. The last part of our testing journey will be to test these Hooks. Routing is an interesting area for testing. We use components from the react-router-dom and this is extensively tested, so there's this sort of literal need to test routing. However, there are some things we should test when routing, so it'd be wrong not to discuss them.

# PART 5 – Snapshot Testing

Snapshot tests are useful for making sure that the UI will not change unexpectedly. An additional Facebook package needs to be added to the project via npm to help us do that. It's called react-test-renderer, and it allows us to render React components as pure JavaScript objects. The advantage of this is we don't need to render it to a DOM so that component can be tested in isolation. If we have an object, we can compare that with a previously rendered object and check that they are the same. To create a snapshot test the process is as follows: Render a UI component. To do this, the create function is use from react-test-renderer. The create function takes an argument of a React component complete with any props. It doesn't use a real DOM, it returns a test render instance, which is a fully rendered component tree as an object, that assertions can be made against. Next, we take a snapshot. React will automatically create a snapshot. On the first run the snapshot file will be created, and stored for future comparisons. In the code, the test renderer instance will be converted to JSON using react-test-renderer's to JSON function. Then we compare snapshots to reference snapshots. On subsequent test runs, the object created will be compared to the stored snapshot. The test should assert that the two are the same. The test will fail if the snapshots don't match. A snapshot may not match, because the change was truly unexpected. Or, if the snapshot has become out of date due to actual changes in the React component. Jest to match snapshot matcher, is used for this assertion. We take the React app as it installs from the create react app. We can edit the app.test.js file. We've moved it into a test folder. We've left the boilerplate test in the test file. It uses another recommended testing utility called React Testing Library. We'll come back and leverage some of its power later, but for now, let's focus on the snapshot test. Just to note, we've chained test with skip, so this test won't run. A report of how many tests we've skipped will be shown on the console line. Skip can be used on any described it or tesco. We've imported the create function from react-test-renderer. That's a package we've added to the project. There's a test feed on line 12, and it's populated with an exe spec. The callback of this spec creates a testing sense of the app component and immediately converts it to JSON. We then assert that the snapshot created by the test will match the snapshot stored. When we start the test for the first time, you can see that all of the tests pass apart from our skipped one, and we have a snapshot written. Looking at the folder structure, we can see a snapshot's file has been added. And inside this, is the app.test.js snap file it's been created. And looking inside that, shows that it's just a text file with the mark up we'd expect. If we press it on the console, it runs the test again. You can see that the test passes now. If we go to the app component and edit the text on line 19. And save this file, the test automatically runs again, and the test now fails. Our snapshot test has failed because the outputted object does not match the previously saved snapshot object. We now have two options. I've replaced the snapshot because we want to keep the updated vendor of the app component. We can do this by pressing U on the console, or the other option is to change the app component back to what it was. We'll do that. The test run again, and you can see that they pass. So, for static components that have no props in those state, snapshot testing is good for doing what Ken said. Using our component in a way that will be used in the app, simply by rendering it.

# PART 6 – Testing Components with Props

Should we test if props or rendered? Yes, we should. Apart from any default props that are supplied, default props are part of the core React code. And as such, their rendering function is already tested. Do we need to test if prop types work? No, this is the concern of the prop types package and tests already exists for this. What we should do is see if the value of props received is rendered, especially if it can change at one time because the component receives a new value for its props. To test the component that has props we're gonna use the test render object again. And find elements in the component that render based on the props. The root of the test renderer can give us an object that allows us to find other elements within the component. We'll use helper methods of the test instance, such as find all by type and find all by props to help us locate other test instances within the three. The children object contains the contents for these instances, and we will assert against them. To show you some testing, we're using that component that we made earlier in the course. And you can see its code now. Note, the class names were given to the JSX expressions returned on lines eight and 14. These will be used in the testing. In the test file, we start by declaring a test suite. This component receives a lot of props, which means lots of tests. So we are grouping similar tests here. As we want to render the same component with the same prompts over and over again, we set some fixtures for the test. We did a components test. This is defined so we can render it in the before reach function. We want to have a stable set of props that we can assert against. So we define an object to use in both props and our assertions. It has the keys and values that meet the component specified prop types. Remember, we're not gonna test the prop types as this is not our direct concern in this component. The before reach function does two things here. It creates a test instance using the create function on the component with props. Note, how we pass in the props object using the spread operator. The second thing we create is a root test instance object. This allows us to find other nodes within this component. The first test that you can see test the h1 render. We find the first h1 and the three using the find by type function. We pass in the element selector is a string. We then assert on this accessing the child array, which in our case is a single text node that should contain the same string I supplied in the header prop key of the props object. There's no need to test this component without the header text prop because this concern of React to core implementation. And we assume that this renders if we don't supply anything. In the second spec in the file, we've had to use a different find function. Unlike HTML's query selector, which returns to the first match, find by type expects to only find one of the specified type in the tree. If it finds more than one, it returns in error. So we have had to use the find all by type function, which returns in array of matches. We know that we are only concerned with the first element in this test. So we've added the element index in square brackets after the call to access the first paragraph. The assertion is as for the previous test, checking that the child value is the same as the supply prop. We mirrored these tests for the next four specs. For paragraphs with indexes one to four, using slightly different matches and values. Note, that we actually called function prop on line 64 to get its value. This is the last spec you can see on the screen. The next number displays render by an array. We may be satisfied if this is rendered as the correct number of child elements while we may wish to check the actual values. That I suppose is the decision for you. We have opted to check each render of each of the paragraphs, failing if just one is wrong. We've also checked each of the renders from the object props display array. We access each array and check the correct value for key has been used along with the correct value attached to that key. You'll notice in both cases, we've used the class name prop of the JSX expression generated in the component to find the element we wish to test. You can access any prop apart from key and ref in the find all by props and find by props functions. Key and ref are reserved by React and throw an error if you try and access them in tests. There's no need to test the render of the unsupplied prop as this is a concern of React itself. And it's supplied with a default anyway. Obviously if we're following test-driven development, we'd want test to fail. We can supply failing values in the second half of our assertions before supplying the passing value if we're using this methodology. So if we run a test, we have, you can see that they all pass. The console gives us options for rerunning tests. A or Enter to run all. F to run only failing tests. And Q to quit back to the terminal line. So now you've seen what you need to test and how to do it when you're thinking about testing components that receive props.

# PART 7 – Mocking Components for Testing

If you think about unit testing that can appear to pose some problems, components inherently render other components that could raise difficulties as we depend on the render of these other components to create the tree. And this doesn't sit well with unit testing. We aren't interested in the rendered components implementation or output, as that should be tested independently. However, to maintain the usability of our component, it needs to re render something when it's called for. Jess has a mock function that will help here. The function takes a string that should be the path to the component to mock. The options allows us to specify a callback that returns a function. The function will return to JSX expression to represent the component we are mocking in the tree. The just mocked functions are written at the top of the test code. Anytime the unit on the test calls for the dependent file, the mock willing to septer and insert the mock code. Our App component now renders component with props and we've run the test for the original snapshot as the reference. The test has failed, that's because the snapshot generated now renders the new version of the App component with the component we props in it. Pressing New Updates to snapshot, and the test will pass again. If we examine the app.test.JS.snapfile, we can see that the file contains the mark of the component with props. Obviously this could cause issues if this component changes in the future. And as it receives props that's highly likely. To get around this, we can mark the component with props in the app.test.JSfile. Anytime it's needed, it will use this mock implementation. And it doesn't matter what the actual component with props does. If that's the case then what we'll do is mock this component using the jest mock function. We'll add the jest mock function now. It's being supplied with the path to the actual component with props file. We provide a callback that returns a function to represent this mock component. We'll allow it to receive props but just return a simple paragraph that contains the text mocked component with props. Once we save this test, it reruns. We'll need to update the snapshot obviously. If we do the that and then check the snapshot we can see that the mock component has been rendered. There's the mock component. So now we've unitized the app component tests and we move the implementation of the component with props from it. We can implement any number of Jess mock functions. So now you can create discrete unit tests for any component that renders other components.

# PART 8 – Testing State Events Interactions

We have stating components. We don't need to test whether the set function works. Is this a concern of the call react code? What we are interested in, is if a component state change is fired by a user or event. And if that event causes the component to re-render with the correct value. Tests usually follow the pattern of arrange, act, assert. In snapshot tests, we don't perform an act part of the test. There are no moving parts. If we have events that cause side effects, like setting state, then we should test that the function attached to the event produces the output we expect. This is particularly useful when testing control components and forms. We can render the form's component and check that the initial value is what is initially set in state. We can then fire the event, say an on change with the value we wish to change it to. Then we assert that the value of the input is what we changed it to. This will check that the event fired, state was set and the element re-rendered with the new controlled value. To help us here, we need a function that will actually trigger the event. This is done by the act function. There are several versions of this function, provided by the different testing packages that essentially do the same thing. But it's crucial that you have the correct one imported. That's usually down to importing the act function from the same packages that the rendering function comes from. The act function has a callback as an argument, and this will trigger whatever you need to and whatever test instance you are testing. Again, the mechanism for actually making the event happen, differs depending on the test package you're using. You can then assert on the value being tested. This pattern can be followed for any interaction within a component. The form component, shown on the right of the screen, is a simple form that has a label for name, and a text input for the user to enter into. State has been declared to hold the value of name. The input calls on change and updates to state. The value of the input is controlled by the state. To test this, we've created a form test file shown on the left of the screen. We use, we act test render's create function to render the tree. We then access the root, which will be the form. And then find the input with the name of name. We assert to verify the initial state of the value set in the component as an empty string. React test render's act function is then called. It takes a callback, which acts on change proper the input, and sets the target objects value to test name. The act function is wrapping the on change. So the callback finishes executing before we move on. When the act function completes, we assert that the value of the input we're testing has been changed to the value we supplied. If we run the test, we can see that it passes. We won't repeat this for any other fields on the form. And we could also use this to test any user interactions that produce other events.

# PART 9 – Mocking Functions

We saw when we were learning about follow-ups that sometimes states is not held on the form itself, but in the parent. To do this, we use them for state to flow to call functions in the parent, by passing them through as props. If we're trying to unit test components, this makes a dependency on the parent, which is not good. Jest provides a way to mock functions in components, and we can leverage this in this situation. This is different to Jest.mark that would be used if it was utility functioning parts from another file. To declare a mock function, we simply declare the mock function's name and set its value to jest.fn. If you need to, you can pass in an implementation call back. This is useful if the function being mocked is supposed to return something. We can then use this when we create the component in the test, passing the mock function name in as a prop. We're not concerned with the implementation of the function being mocked, because it's not declared in this component. We're happy if our component and the test calls this function with the correct arguments. Jest provides matches for function calls. The mains ones to know about are to have been called which returns true if the mock function was called, and to have been called with, which takes a set out of arguments and returns true if the function has been called with the prescribed arguments. If you're unit testing functions, the other matches might be useful. Let's see how we can implement this in a component test. We modified the form so it has a submit button and a non submit event handler. The phone function will call submit from props with the value of state, in the test we will mock the submit function passed in by props and use this when we render the form component. We'll simulate a click on the submit button. This should trigger the unsubmit event handler which in turn will trigger the submit function. There's a small hiccup here, react test render doesn't allow us to fire events without the event handler being defined. So we'll need to use a different library. We said earlier that we'd leveraged from the power of react testing library, so here it goes. We are testing libraries paid into our project, so we don't need to install it as a dependency. In our test suite for the form, we declare mock submit as a Jest function. We have also set a const for the test name. We've changed the test for the unchange to work as a test in react testing library. This involves creating a container using the render function, similar to using the create function to make a test instance in react test renderer. We then find the input in the container using the query selector function and its CSS selector and check its value as an empty string. We're testing library has an enhanced user events, so we've leveraged this here to type in the inputs. It simulates typing tests name into the input, firing it's on change event and updating the state. We then assert that the value in the name input is the value that was typed, and this is the same as the test that we did earlier. That's the mocking of functions now we can assume on submit event is fired calling the handler submit function, and ultimately the submit function from props. And if we can record the calling of the submit function from props, and what it was called with, we can make assertions, setting a value in the name input and firing a click event on the submit button should do this, so that's how we test. In the second spec, we get the container, the name input, and the submit button. We perform the actions of typing in the name input and clicking on the button. We use fire event from react testing library here, instructing it to click on the identify button. We then assert that the mock submit function has been called exactly once, checking this is good practice because it shows your functions work correctly. We also check that it has been called with the correct document, there'll be an object with a key of name and a value of tests name. Running these tests show that they pass. You can use this function mocking anywhere you have a dependency on a function that comes through props. So with this, you should be able to test the components that have their own internal data or anything that's passed to them by props.

# PART 10 – Testing Components Asynchronously

Now we're going to find out about how to test components that use asynchronous data. To fully test components, we need to know that they make the right asynchronous calls, that they respond appropriately to data or errors that are returned. And conditional rendering works. This is probably the most difficult aspects to testing [React](https://app.qa.com/learning-paths/reactjs-1377/) components. There are so many moving parts and time delays to deal with.

One way this can be made easier is to use a third party package to handle asynchronous calls. This means the whole module can be marked. One such module we could use is called axios. Axios is commonly used. In fact, the jest documentation shows examples of mocking responses with it. Basically, it simplifies HTTP requests by having functions that are the same name as the verbs we use.

The definite advantage is that it automatically throws an error if a 200 status is not received, meaning only good data pass through to set our state. It also puts the data straight into a data key in the response object removing the need to await the call to JSON.

Before we get into the testing, we're showing you a component that has a fetch call made with axios. There's a similar pattern to what we discussed in an earlier course, we have a use effect hook that calls an asynchronous function get data. This is where the axios call is made. Line nine if you haven't spotted it yet, like fetch for get requests, we just pass in the URL for the request.

For the calls like posts, we can configure an object like we did for fetch. You can see its error handling is much simpler than fetch calls were made before. We surround it in a try catch blocks. Dealing with the error object generated by axios and handling the response status is here, it's much cleaner. If you look at the rendering of the JSX elements we return in this component, you'll notice that there are data test ID attributes attached to the elements. These are there so we can identify the components for testing. There'll be included in your production code. And there is a babel package that can be used to remove them. But that's not really necessary.

We'll move on to the testing now. We're going to test the asynchronous calls axios makes, we should provide a mock for it. We've created a folder called mocks with a double underscores around them, and a file called axios, which you can see in the file explorer on the left of the screen. In this file, we export an object. This contains the names of the functions we want to mark as keys in the object. The values are what we want to provide as the mock functionality. In this instance, we might get as a jest function and have it marked resolved value of an object with an empty data key.

If we wanted to mock more axios function say post, we would have the mock implementation here, adding a key of post, and whatever we wanted to add. As we have this folder and jest has been set up through create react app to look for mocks before performing actual imports. This will be used whenever we import axios in our tests. This is not well documented, so you do well to remember it. The actual test file for the component is set up in the same way we have set up for the tests. We'll discuss the imports from react testing library as we need to, and the jest don't extend expect so we can use some additional matches.

Notice that we import axios mock from axios. We alias our function just to be explicit that we have mocked the module. Remember that's in our mocks folder, line seven called afterEach with cleanup. Cleanup is a utility function from react testing library that removes a tree rendered as part of a test. It's good practice to do that.

The first spec in the file is to test the fetching and displaying of data. Note that its async as we're dealing with asynchronous actions. Line 11 makes a call to axiosMock.get And we override the mock resolve value with mock resolve value once and a value to return, they should reflect the data structure that the endpoint returns and it will replace the call in the components. We then render the component destructuring the get by test ID, and get all by test ID functions to use.

To ensure that the loading message is initially rendered, we use its test ID attribute of loading and assert that the element has text content of loading data. The next part initiates the asynchronous use effect call. We make an array of Li elements that have resolved. This is done awaiting the wait for function and supplying a call back to return any element has a test ID have resolved. The wait for function is part of react testing library. It will wait for the DOM to become stable before moving on.

We then assert that there are the correct number of allies rendered, in this case two, and we check that the access mop get function has been called exactly once, a check that those cyclic requests are not being made. At the time of recording, we have to update react testing library to version 10. And also installed and updated the jest environment JSDOM 16 to use these features. We updated the other testing library packages for adventure two.

It's worth noting that create react app is not always up to date with the latest packages. And searching online can often help with suggestions if things appear to go wrong. We've also tested for errors for making the axios get function, reject its promise with particular data. The system with a simple error object in the case of a network error, we follow the same pattern as for the resolved promise. But this time look for an error element with a test ID of network error. And for it to have the text content that's the same as is rendered in the component.

For the response errors, we create an error object that has a response object with status and status text keys, and use this in the promise reject. We check for an element within a test ID of response error, and then assert that the correct message is displayed. As a reminder, at the time of recording, we have to update react testing library to version 10. And also install an update of the jest environment JSDOM 16 to use these features.

So now you can test asynchronous calls and components using react testing library, mocking the package that makes the calls and checking data and errors dealt with appropriately.

# PART 11 – Testing Components with Routing

Routing testing is not done often, as the package is extensively tested by its creators. Issues arrive when you have components that use router components such as link or route as part of their rendering. They use context and more produced testing errors if not dealt with properly, especially if they use router features outside general navigation. For example, a link that expands a list inside a component.

It may be tempting to stop the components yourself to get around this, but the creators of react-router-dom suggest surrounding your unit test in a router component. The most appropriate is the memory router, as you're able to reset the router between tests. The memory router allows you to put an app at any location using the initial entries and initial index props.

Navigation and checking locations doesn't really need to be tested either, but there's some information about testing these in the router documentation and [React](https://app.qa.com/learning-paths/reactjs-1377/) testing library documentation, if you really feel you have to do this. With that said, there's nothing else really to show you here. Testing snapshots and component outputs that render routing components can be done without the need to mock or stop the router components, or indeed, surround your test in a router. They're not included in the snapshots made, and don't generally affect all the tests.

# PART 12 – Testing Custom Hooks

The final part of our testing journey is to look at testing custom hooks. Testing hook calls in components can be done using the jest.fn call and providing an implementation to return anything the hook may provide. That could be a dispatch function if you were using a user reducer hook somewhere or a value if you are using a use context hook. But that doesn't really help with testing the functionality of a custom hook. Hooks can only be called from inside the body of a function component.

One solution may be to write a component for testing the hooking, but that seems like a lot of effort, as you'd have to trigger all the hook updates inside that component. A React hooks testing library package helps create a testing environment that can handle hooks without you having to write a component for it to be tested in. It lets you use the hook directly and assert on what you should've done. It's written by the same team that wrote [React](https://app.qa.com/learning-paths/reactjs-1377/) testing library.

There are a couple of caveats for using this hook, and they can be found in this documentation. They suggest using the library when you have one or more custom hooks that are not tied directly to a component or if you have a complex hook that would be hard to test fire a component. They also suggest that you shouldn't use the library if the hook is defined and used only in a single component, or the hook can be fully tested by the components that use it.

To demonstrate this, I'll talk through an example similar to the one given in the React hooks testing library documentation. You can see that we have a use-counter hook on the right hand side of the screen. This will be called in a component and deals with the state and updating of count having a function called increment. This function, through the use call back hook from React, simply takes the current state adds one to it and sets the new state. The initial value of state can be passed to the hook as an argument.

In the test file on the left we've installed the React hooks testing library and imported render hook from it. This is the function that allows us to use the hook in the test without rendering a component. We've also imported act to allow us to effect stating the hook. We have a test to check that the increment function in the use counter hook will actually increase the value of state. We deconstruct this hook from the return of render hook to which we pass a function. That returns our use counter hook setting count initially to 100.

We then wrap a quarter increment on results current property in the act function. This call accesses the current value of result and executes increment. Final part is the assertioner. Having guiding one to the initial count for the increment function. The current value of count again access free result.current will be the value of 101.