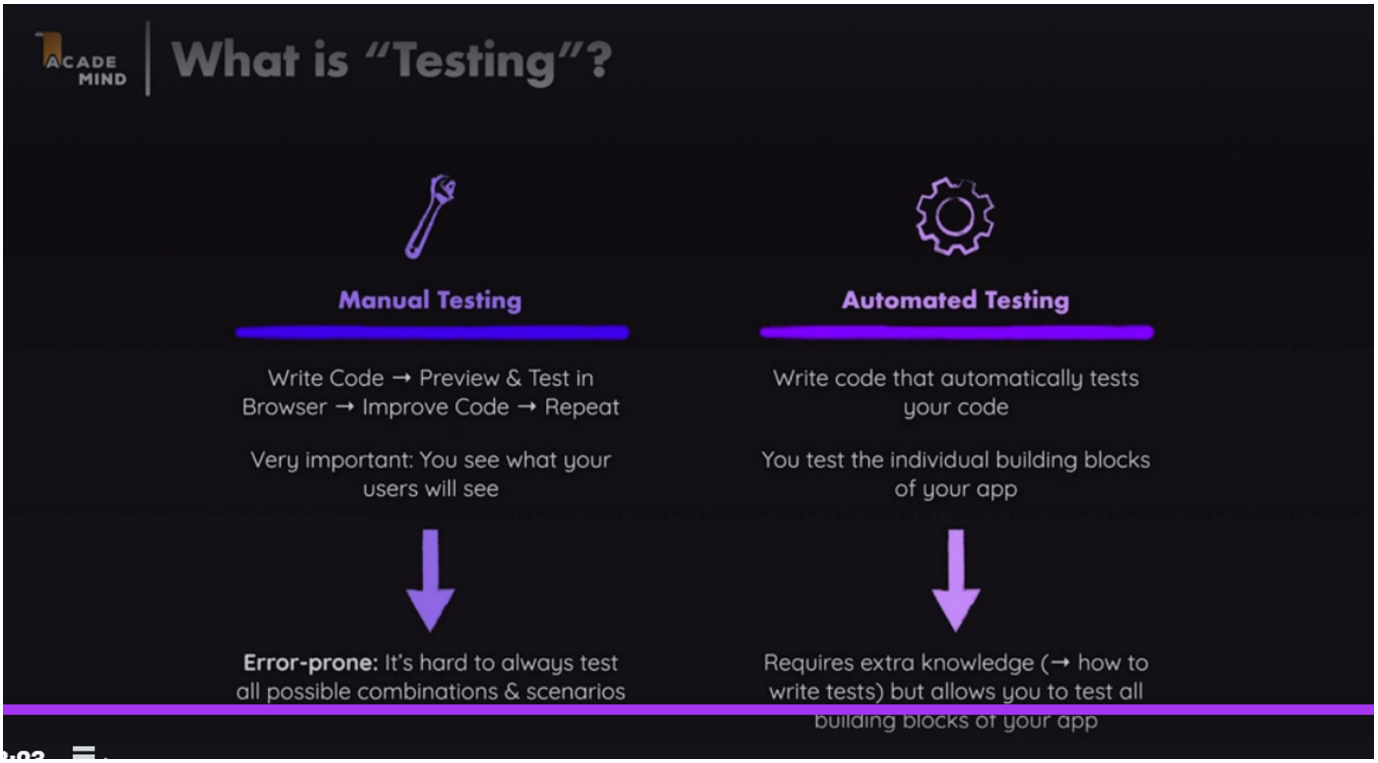
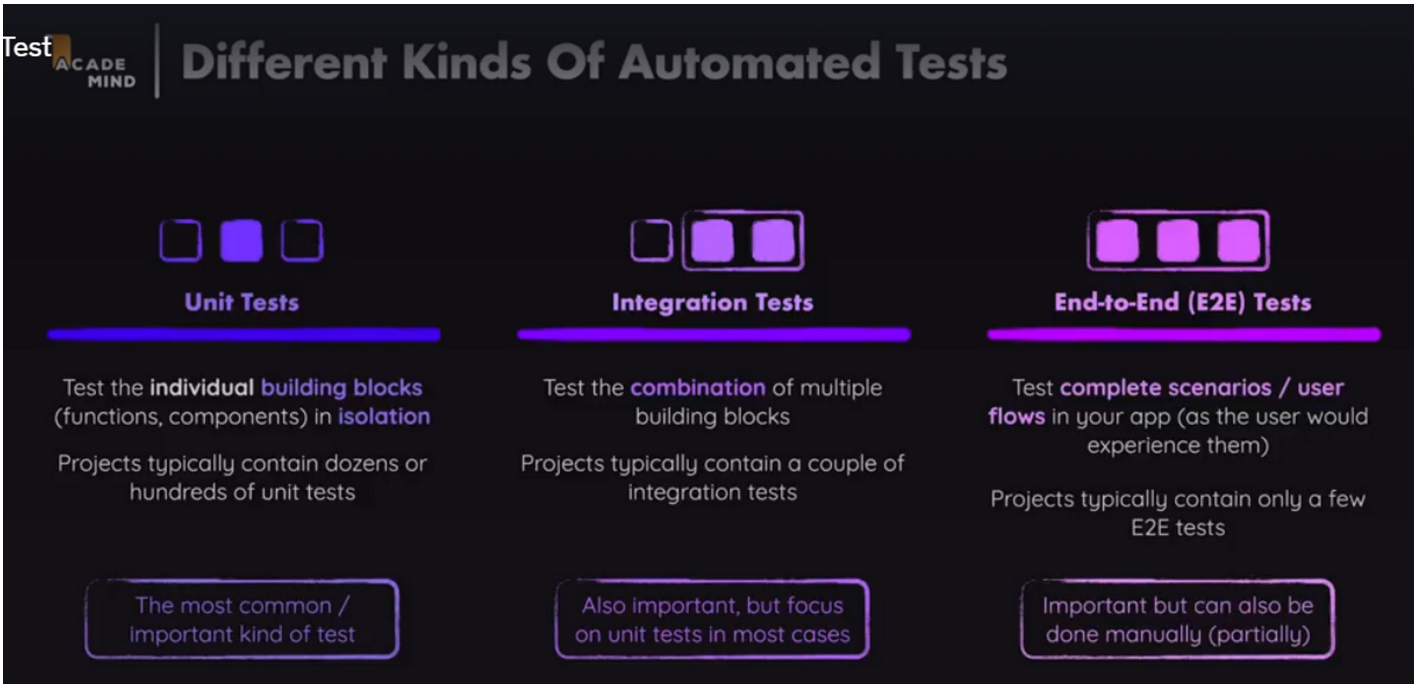
Section 29: Testing React Apps (Unit Tests)

Thursday, October 3, 2024 1:43 PM

What and Why?



Understanding Different Kinds Of Tests



So before we get started writing any code, what actually is testing?
Well, we did test a lot in this course already, we did a lot of manual testing. So that means that we, as a developer, write some code to implement a certain feature or fix a certain issue and then of course, we preview that application in the browser and we test it there. That's what we do all the time as developers, that's what we did all the time in this course, it's super important because we see what our users will see and we definitely wanna polish that and make sure that this works, and that this is not going to go away. But, testing the app manually is also error-prone, at least if it's the only kind of testing you do. Simply because it's hard to test all possible combinations and scenarios. If you have a very complex React app with a lot of different pages and features, and you then add a new feature, or you change an aspect of an existing feature, you will probably test that change or that new feature, but you're not going to test all the other parts of your application all the time. And therefore, you might change something which breaks some other feature in your app and since you're not testing everything all the time, that breaking change, that bug, might slip through and maybe you eventually catch it later, but that's not guaranteed and you might catch it too late and it might then cause some extra work.
And that's where automated testing steps in. This is not a replacement for manual testing, manual testing will always be super important, but it's an

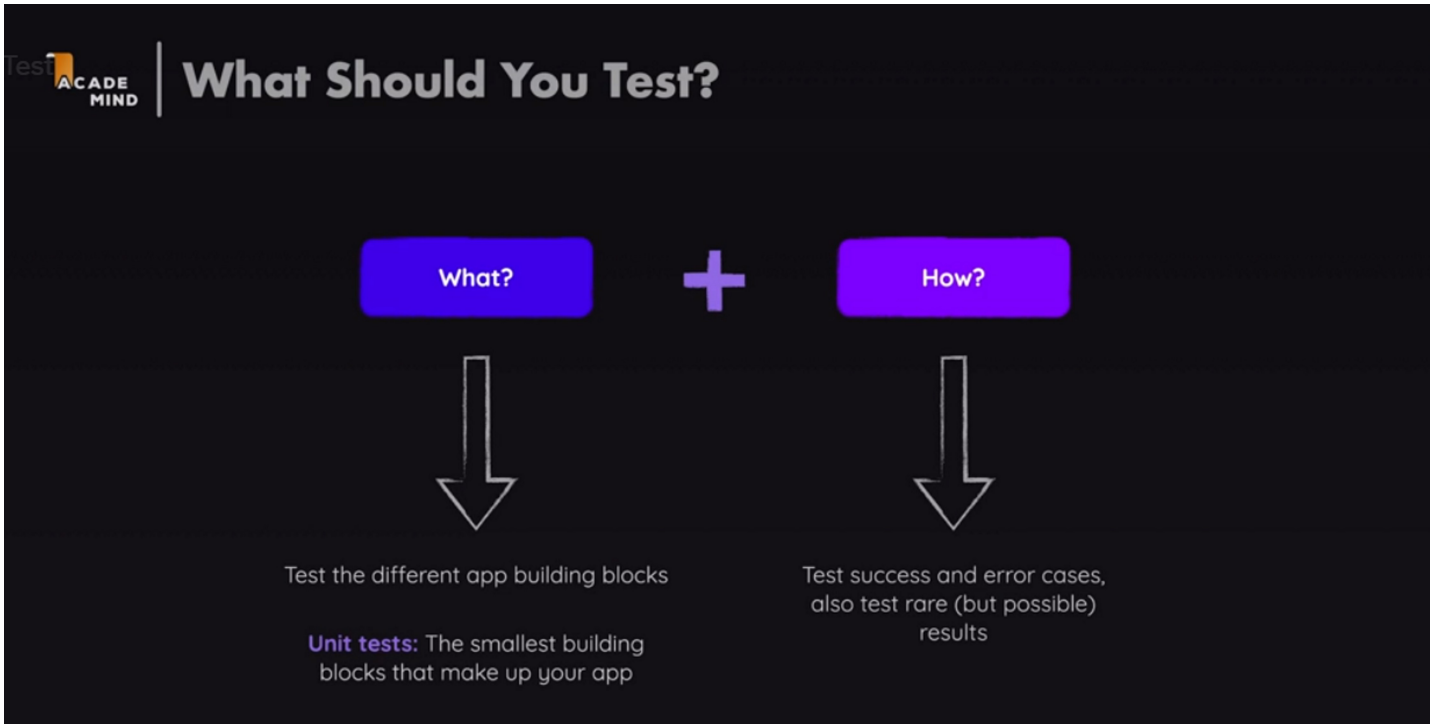
addition. With automated testing, you write extra code that runs and tests your other code, your main application code. And this might sound super weird, but this is a standard procedure, a standard thing to do in modern development. And the good thing about that is, that since you write code that tests your entire application automatically, you can always test everything no matter what you changed because you will automatically then test your entire application, it doesn't cost you a lot of time to do that. And therefore, you typically write such tests, such automated tests that test the different individual building blocks of your app, but where you then test all those building blocks together, whenever you make changes to your code basically, instead of testing only parts of your app from time to time. And that's therefore very technical, but it allows you to test everything all the time and combined with manual testing, that allows you to catch errors way earlier and it allows you to write and ship better code in applications.

Now, when we speak about automated testing, it is important to understand that there are different kinds of automated tests which you can have a look at. For example, there are unit tests and I can already say that those will be super important. But you also have so-called integration tests and end-to-end tests. Now, these are the three main categories of tests which you can write. Unit tests are all about writing tests for the smallest possible units of your application. So for functions, individual functions that you use in your application. Or in the case of React apps, testing some components independently from the other components in your app. And therefore, projects typically contain a lot of unit tests, since you basically wanna test all the units, all the functions and components that make up your application. And hence this is the most common and important kind of test. And the idea simply is that if you test all the individual units on themselves, the overall application will also work. But to verify that the overall application really works if you put together all those units, we have integration tests. Here, we test the combination of multiple building blocks. For example, multiple components working together. And projects typically contain a couple of such integration tests, but not as many as you have unit tests. Though as you will also see throughout this course section, it's not always that easy to differentiate between unit and integration tests when testing React apps, since very often, you wanna test a component that also uses some other component. But we're going to see that in action throughout this course section. So generally, integration tests are also extremely important, but we have fewer integration tests than we have unit

tests.
And then we also have end-to-end tests. These are all about testing entire workflows in your application, you could say. Entire scenarios, like logging a user in and then going to a certain page. So these aim to really reproduce what a real human would do with your website. Basically what you would also do with manual testing, just automated. Now, whilst this might sound like the most important test, it definitely is important and you therefore typically do write some end-to-end tests, but not as many as you have unit and integration tests, because if your unit and integration tests work, you can be pretty sure that your overall application works. And then those unit and integration tests are simply easier to test. They often are quicker to run and they are more focused and it's way easier to test all possible scenarios, if you test all your units for different scenarios then coming up with all possible scenarios, if you look at your app as a whole, which is what you do typically with end-to-end tests. So these are important, but it's basically all to what you do manually and therefore you have fewer of those tests.
Now in this course section, since it's an introduction to testing, we will focus on those most important kinds of tests: unit tests and also to some extent integration tests. But at the end of the module, I will also point you at some other resources, which you can check out to dive deeper into React testing if you want to.

What To Test & How to Test

Now, when you start writing tests, there are two super important questions you have to answer, relatively early. And, these questions are what you should test,
and how you should test that.

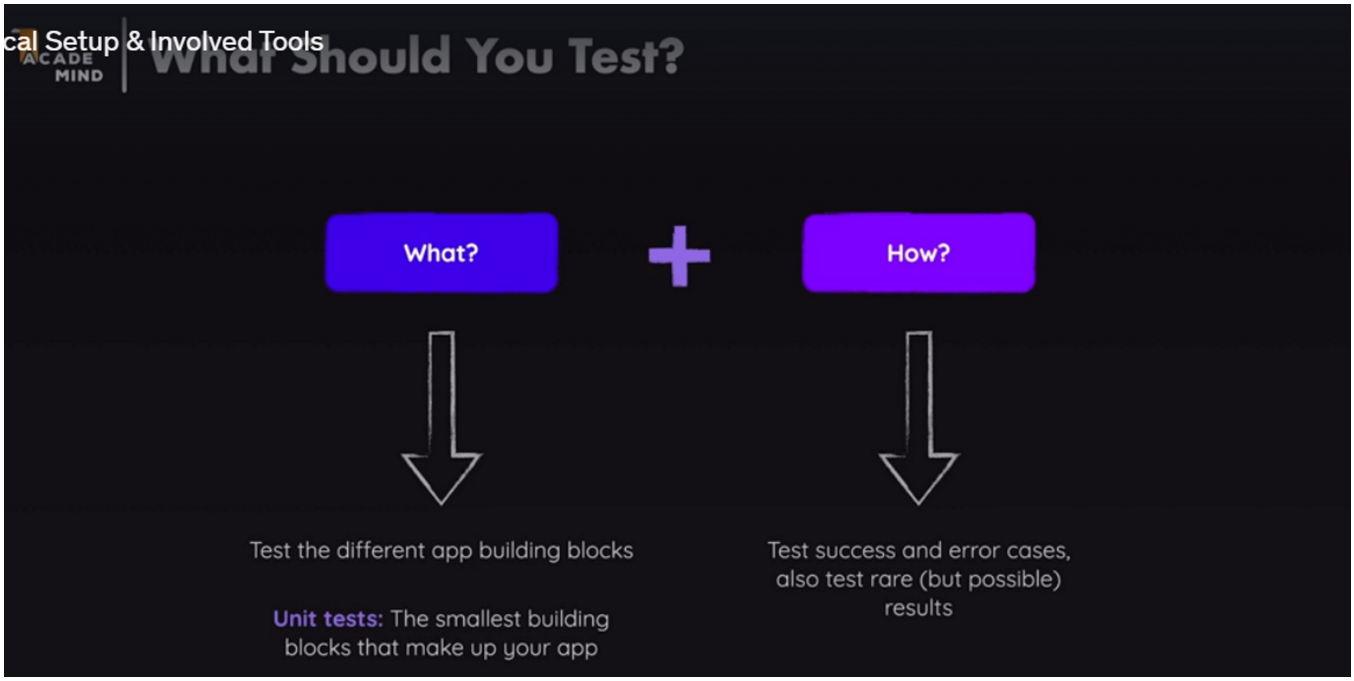


I don't mean how you should technically test that—like how you make sure that you have some code that is then executed by some other tool. I'll come back to that technical question later, but I mean which kind of code should you put into your testing code.

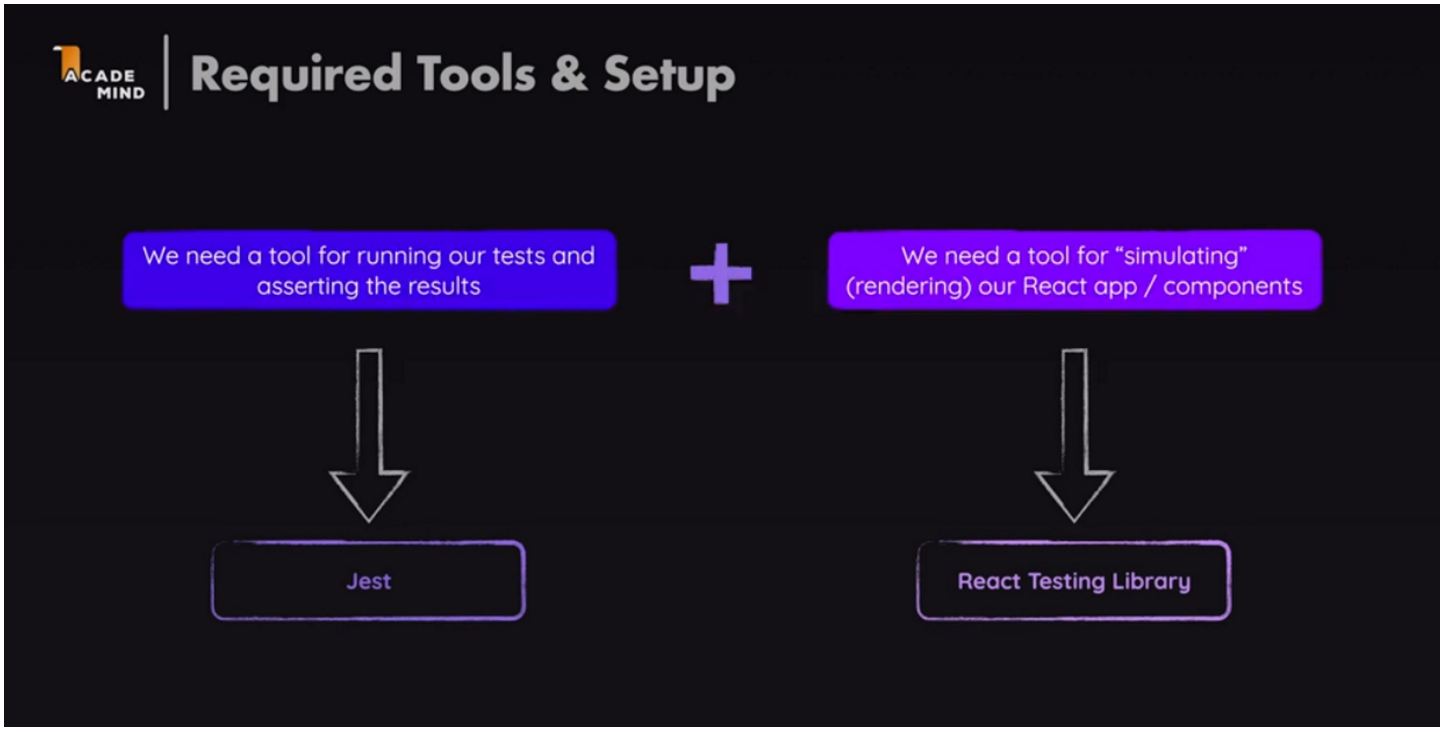
When it comes to what to test, as mentioned for unit tests, you want to test the different building blocks that make up your application, and you really want to test small building blocks so that you have small, focused tests that only test one main thing each. This way, you have a lot of focused tests that fail for a clear reason if they do fail, instead of having a few large tests, which could fail for all kinds of reasons.

Now regarding the "how," you want to test success and error cases that could occur if a user interacts with your application. You also want to test some rare but possible scenarios and results. Again, that will become clearer once we start writing some code.

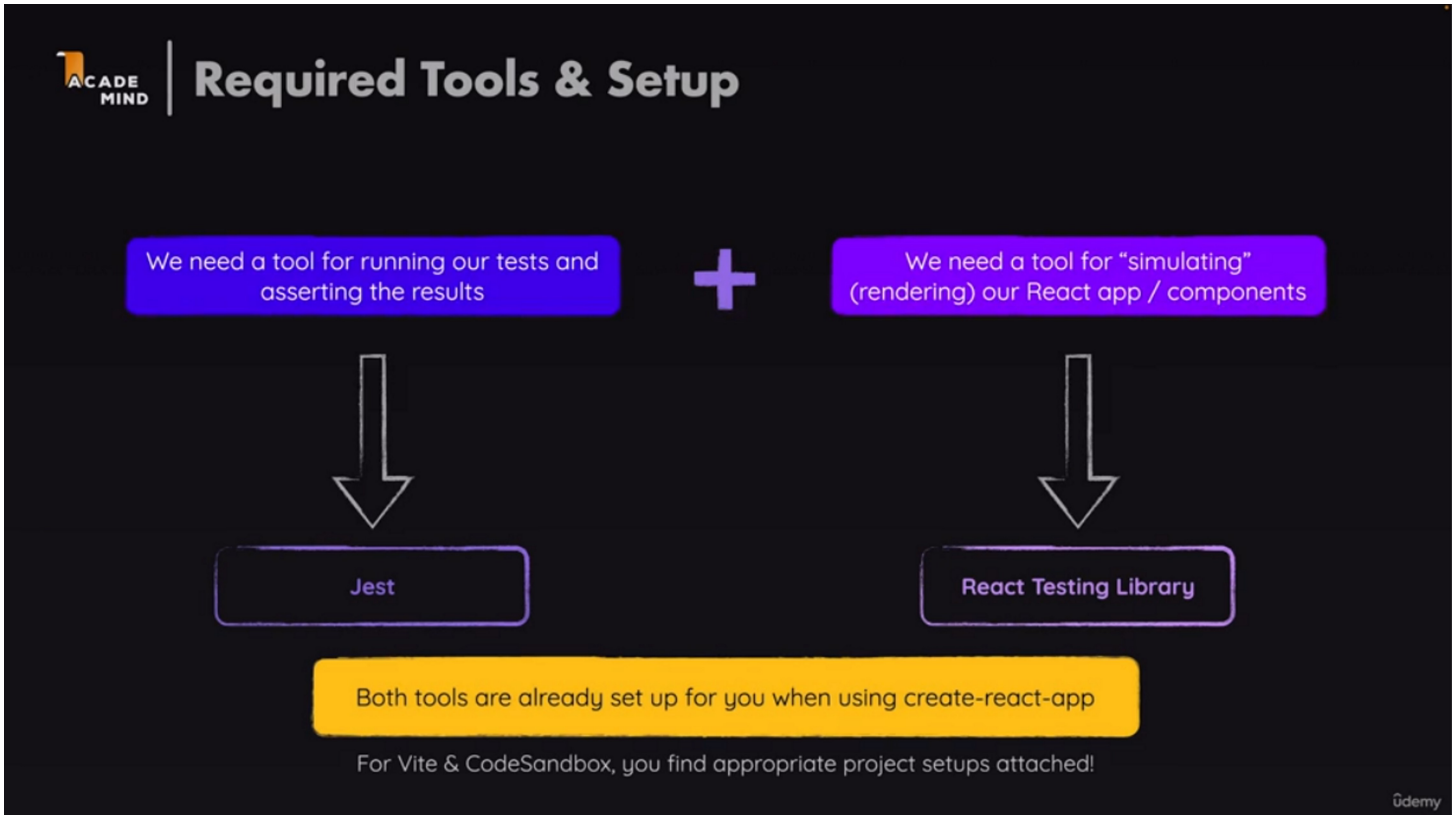
Understanding the Technical Setup & Involved



There is another important question: we now have a rough idea of why we need tests, and we know that tests are code that then tests our **application code**, but where do we write this code and how do we execute this testing code? For this, we need some extra tools and an extra setup.

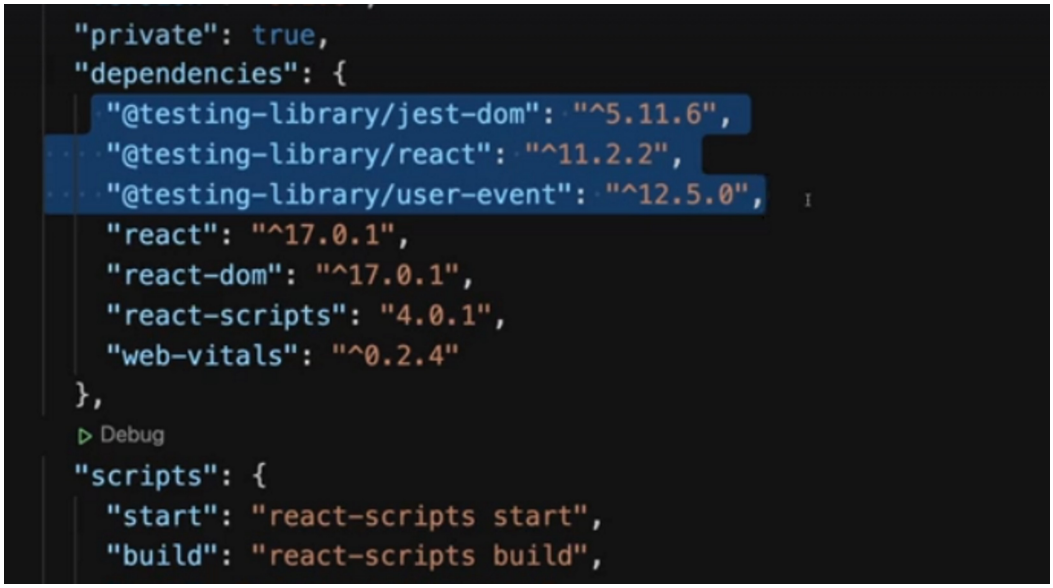


And specifically, we need a tool for running our testing code and for asserting the results. To determine whether some result can be seen as a success, or if a test failed with a given result, we need a tool that does that. In our React app, we also need a way of simulating the rendering of our React app and components for those automated tests to interact with them. So we need to simulate the browser, so to speak.
For the first part—running our testing code and asserting the results—we typically use **Jest**. It's not the only tool for this job, but it's very popular, especially for React, and it's easy to use. For simulating and rendering our components and the React app, we typically use the **React Testing Library** these days.



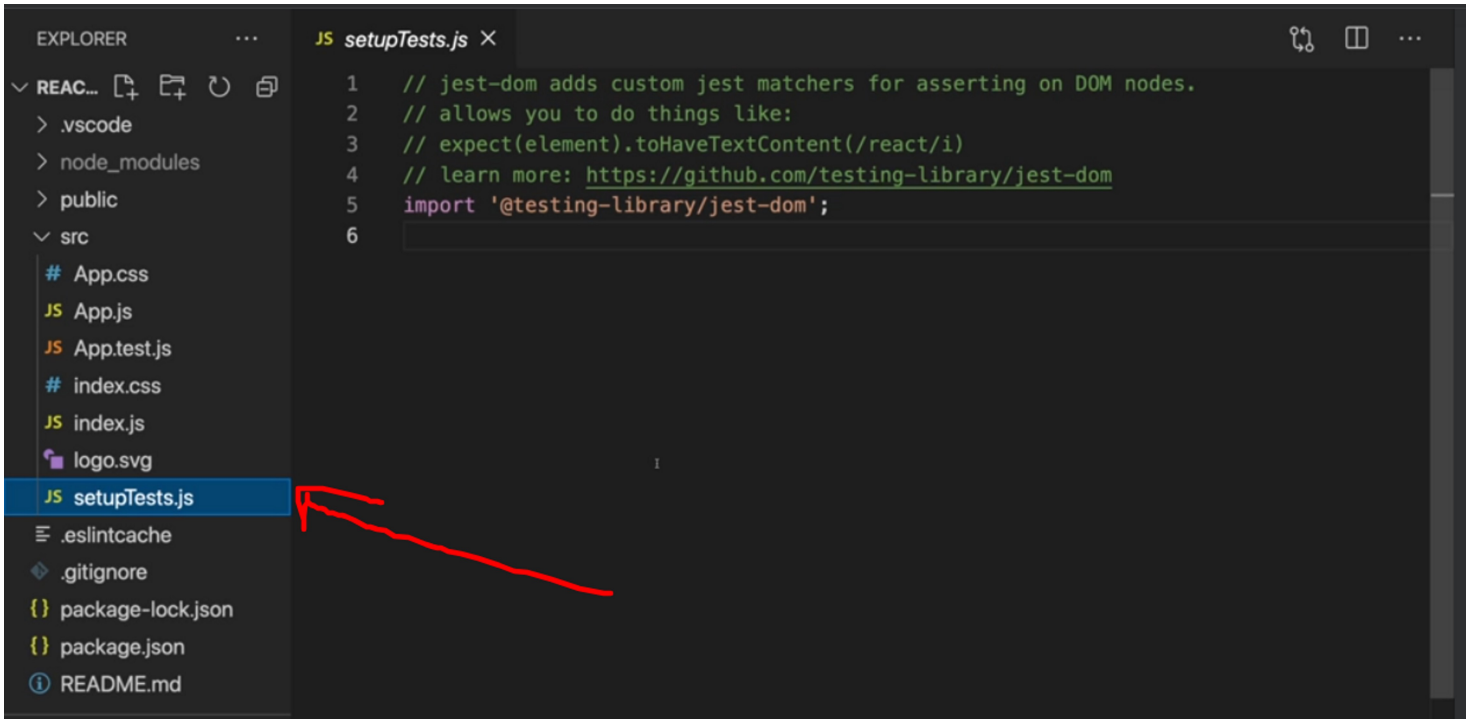
And both these tools are already installed and set up for you, when you work in a project created with 'create react app'.

If we have a look at **package.json**, we see those Testing Library packages, which are part of this project. I did not install those—they were part of a newly created React project, created with **Create React App** automatically, out of the box.
Now, what we don't see here is **Jest**. We only see the **jest-dom Testing Library** subpackage, but I can tell you that it's also part of this project, just as a dependency of one of those other dependencies, basically. Therefore, in a brand new project like the one attached, everything is already set up for you to run your tests.

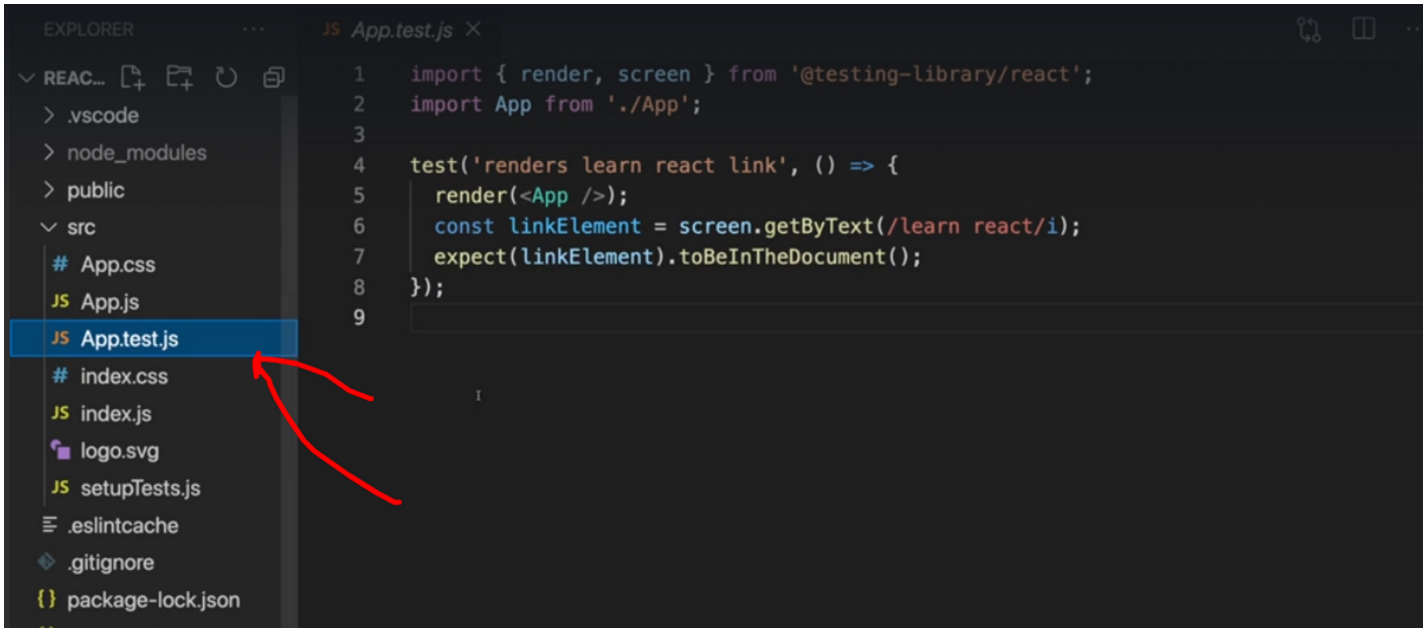


Running a First Test

setup.js file as the name implies, does some setup work. We don't need to do anything else with this file.



App.test.js file tests our App Component.

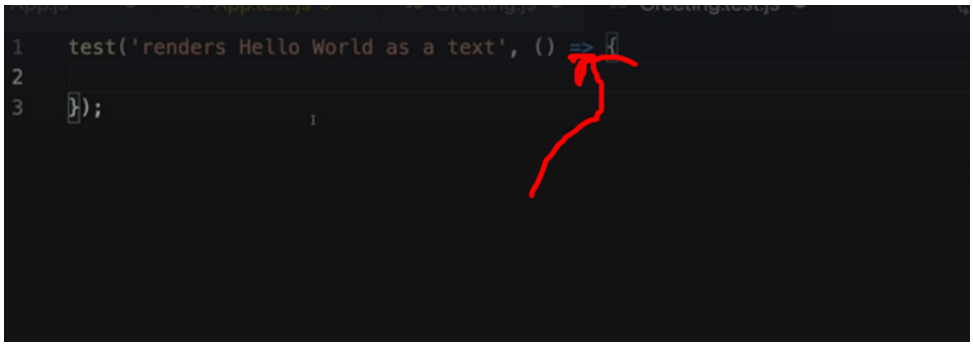


To run test

npm run test

Writing Our First Test

Then we need to add a second argument to this test function, which is an anonymous function, which will contain the actual testing code.



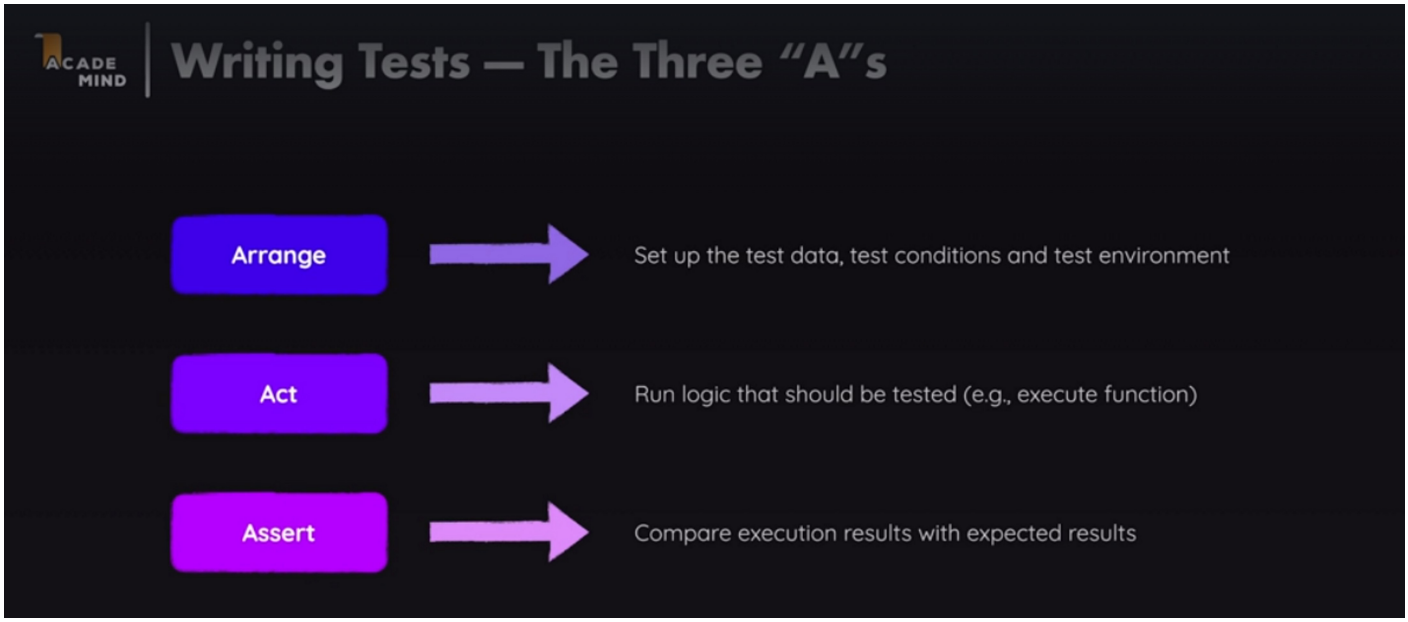
Now, in here, we typically want to do three things when we write a test. We want to write a test by using the three A's.

The first A stands for arrange. We want to set up our tests. For example, we want to render the component we want to test. We can also do additional setup work if required.

Then we want to act. We want to do the thing we actually want to test. For example, if we want to simulate a button click, we do that as a second step. It's not something we do here, but it is something you'll often do in some tests, and it's something we will also do later.

Last but not least, we want to assert the results. We want to look at the output visible in the browser, for example, and see if it matches our expectations.

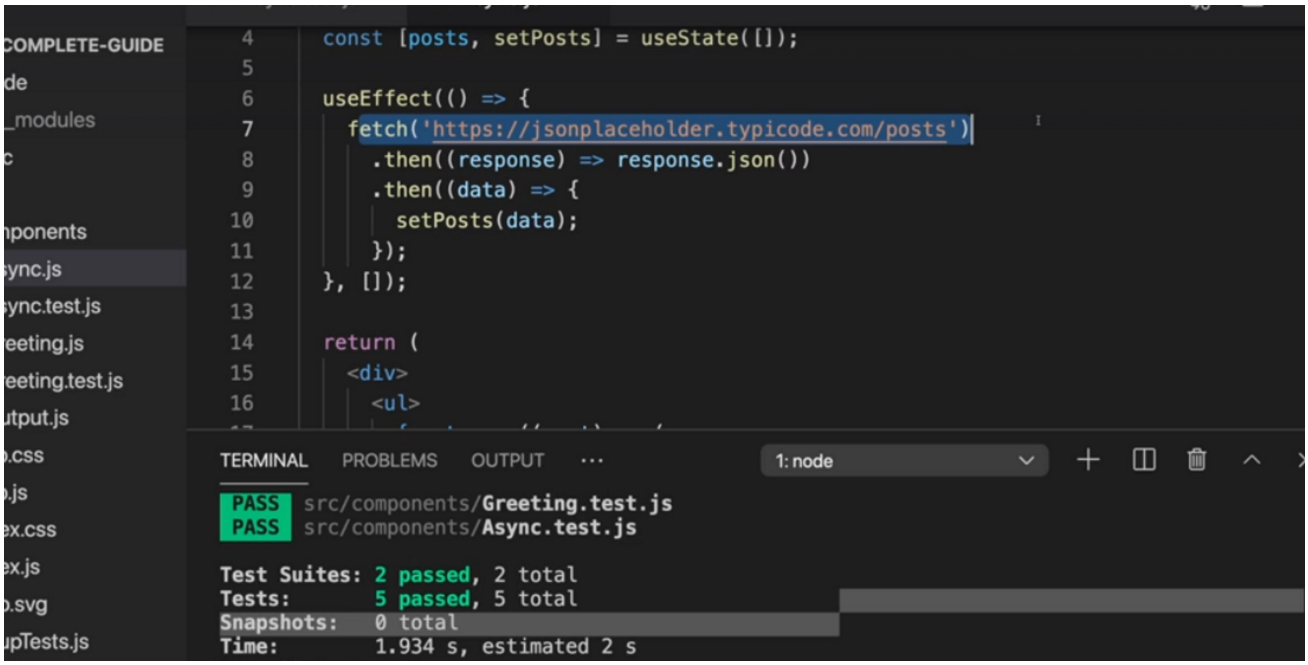
So these are our three A's.



Testing Asynchronous Code

https://www.w3.org/TR/html-aria/#docconformance

Working With Mocks



You don't want to send requests to servers that start changing things there. So, when we write tests, we either don’t send a real request or send it to some fake server, some testing server. Both are viable approaches, and I'll go for the approach where we don’t even send the request in the first place.

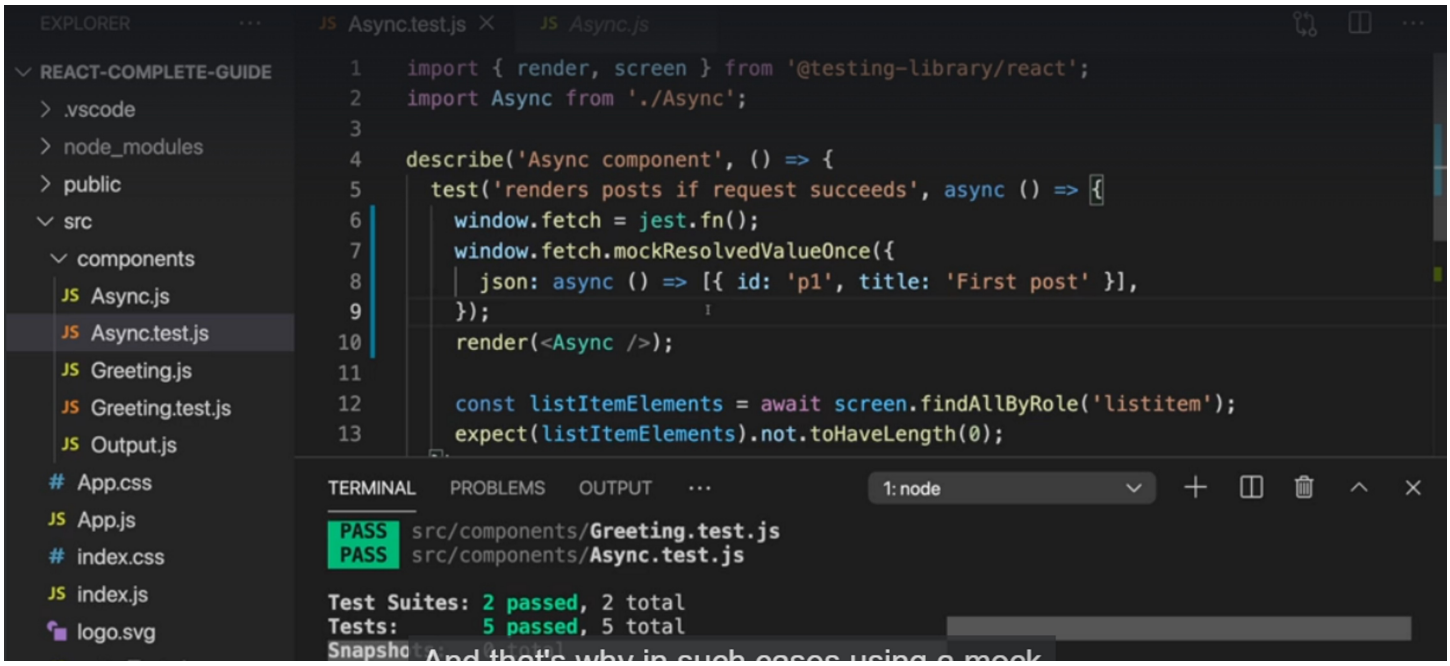
There’s one super important thing to note: when you write a test, you don’t want to test code you haven’t written. In this case, I don’t want to test whether the fetch function works correctly and sends a request. The fetch function wasn't written by me—it's built into the browser. I rely on the browser vendors to have written that function correctly.

So, I don’t want to test whether fetch successfully sends a request behind the scenes. Instead, I want to test if my component behaves correctly depending on the different outcomes of sending a request. I want to check if my component behaves correctly once I get the response data, and I also want to check if my component behaves correctly in the event of an error. But I don’t want to check whether sending the request technically succeeds.

Therefore, we want to replace the fetch function, which is built into the browser, with a mock function—a dummy function that overrides the built-in function. A dummy that does what we want without sending a real request. So, when our component executes during testing, we use that dummy function (the mock) instead of the real built-in function.

That’s what I want to do here. This is such a common scenario, not just for fetch, but also for working with localStorage, for example, where you also don’t want to trigger changes in the actual storage.

This is such a common scenario that Jest, the testing tool we’re using, has built-in support for mocking such functions, making it fairly easy to do.



We are simulating this success case, and we're not actually sending a request to the API. Therefore, we're not hammering that API or sending unnecessary requests. We are reducing the amount of network traffic, and we also avoid potential problems if the server is down and our tests would fail for that reason, which of course is not something we want. We can control different outcomes for this fetch function to test different scenarios with our tests here. That’s why, in such cases, using a mock is not a bad idea.

Summary & Further Resources

https://jestjs.io/

https://testing-library.com/docs/react-testing-library/intro/

https://www.w3.org/TR/html-aria/#docconformance

For testing React Custom Hooks (Custom Hooks)

You should test your entire code base if possible.

https://github.com/testing-library/react-hooks-testing-library