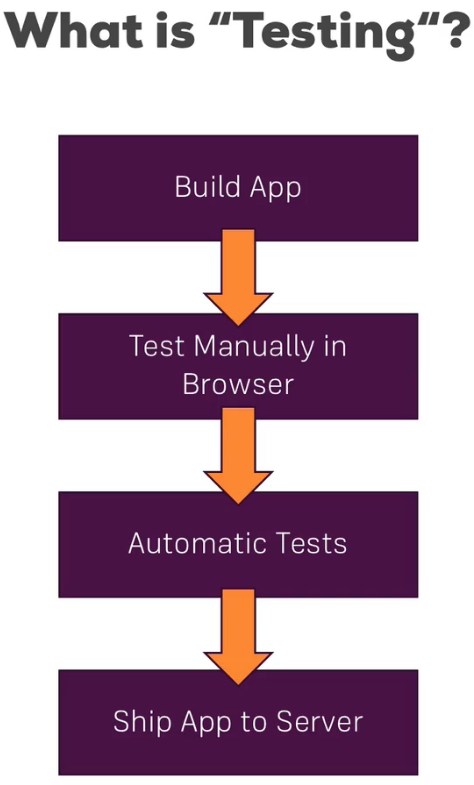
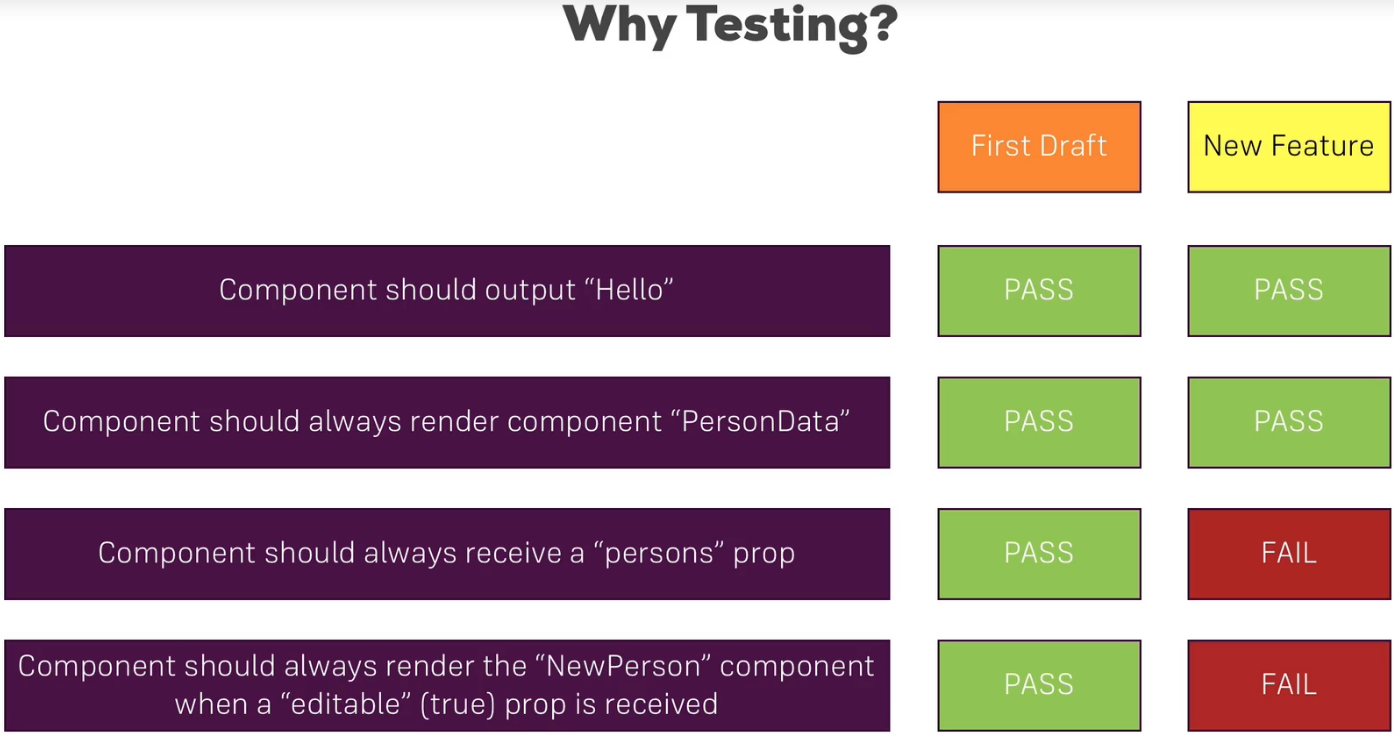
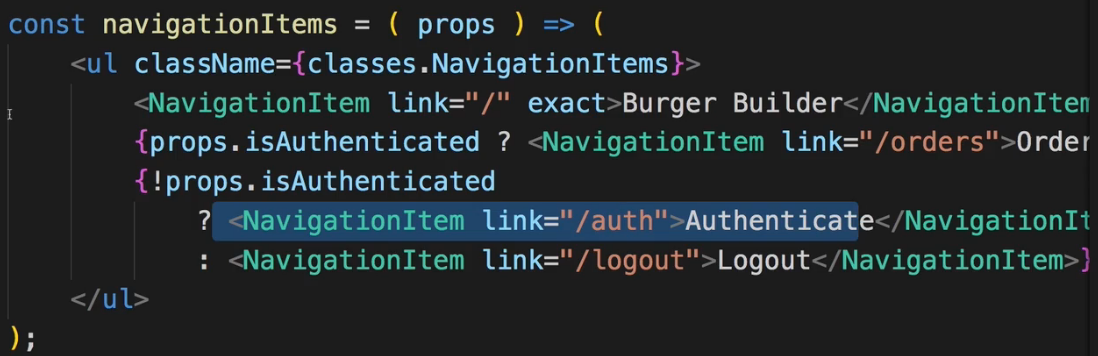
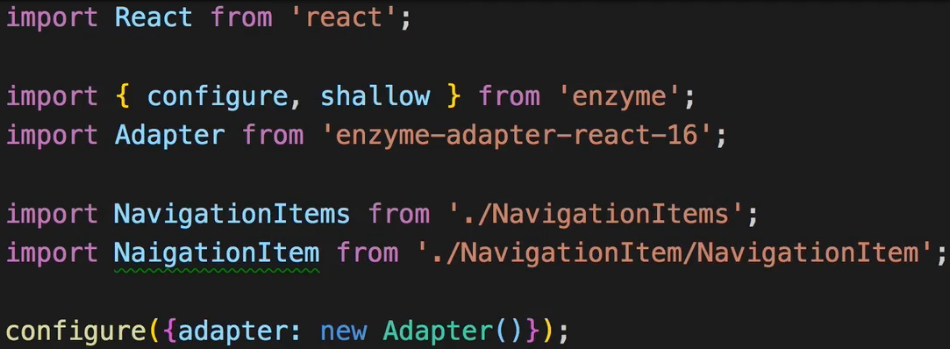
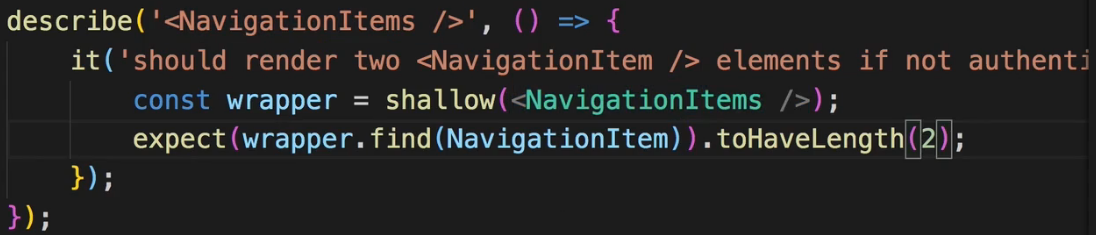
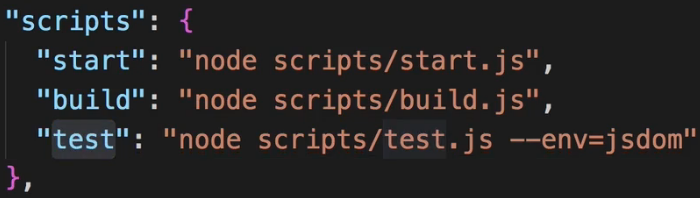
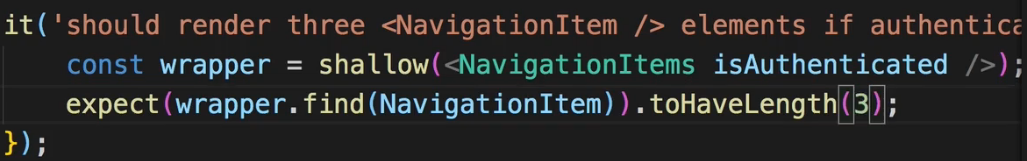
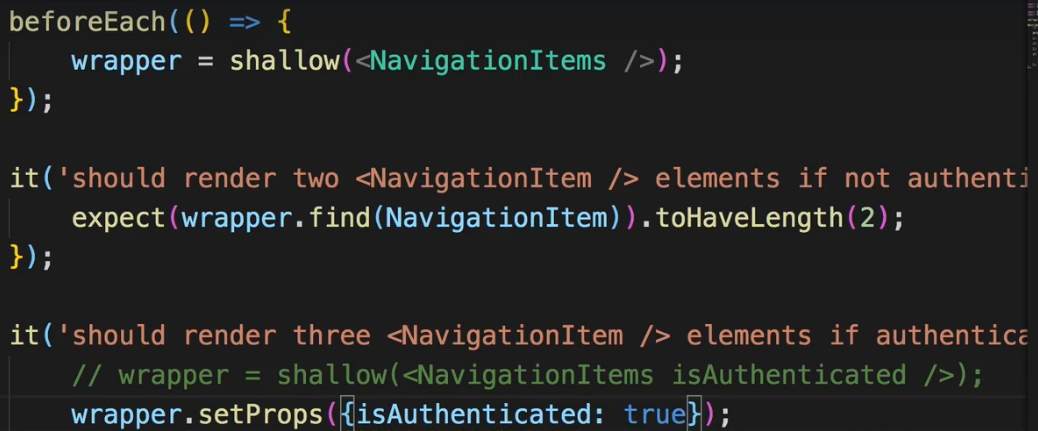
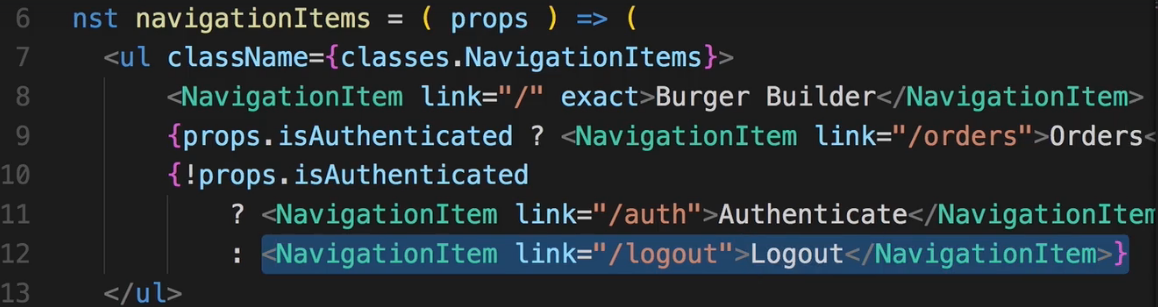
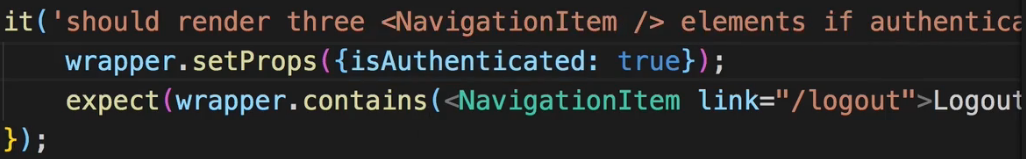
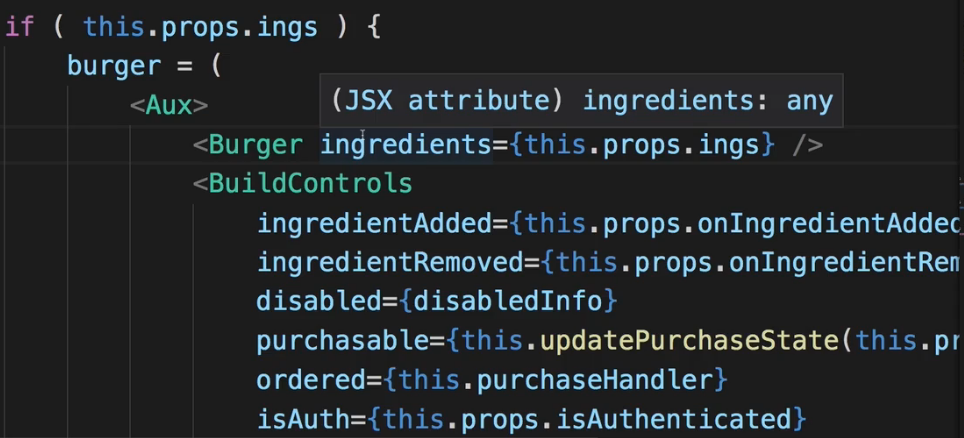
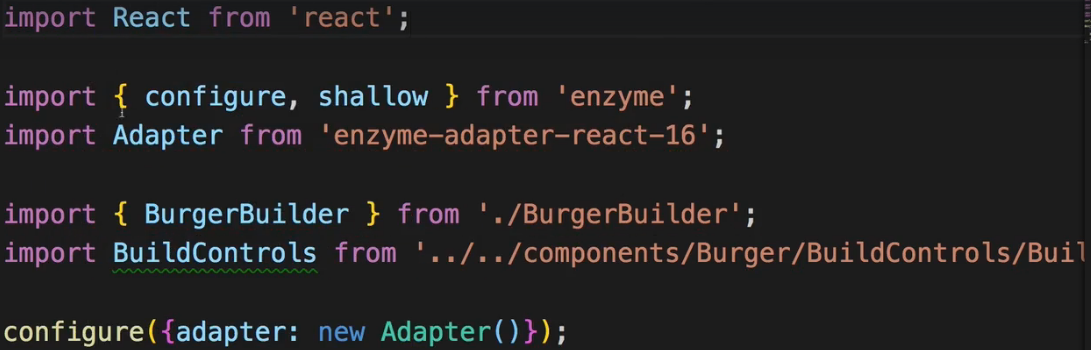
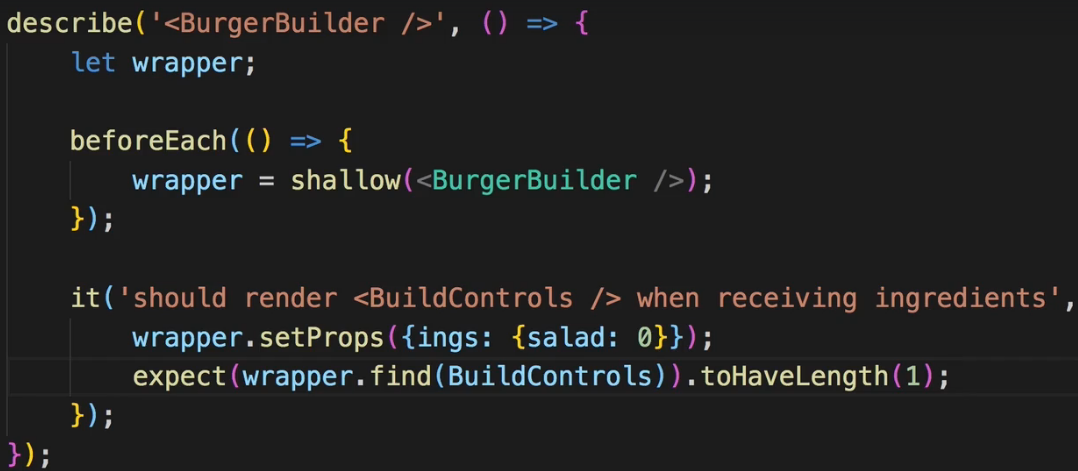
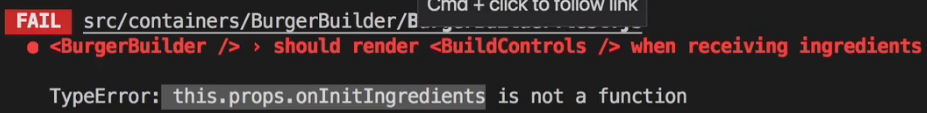
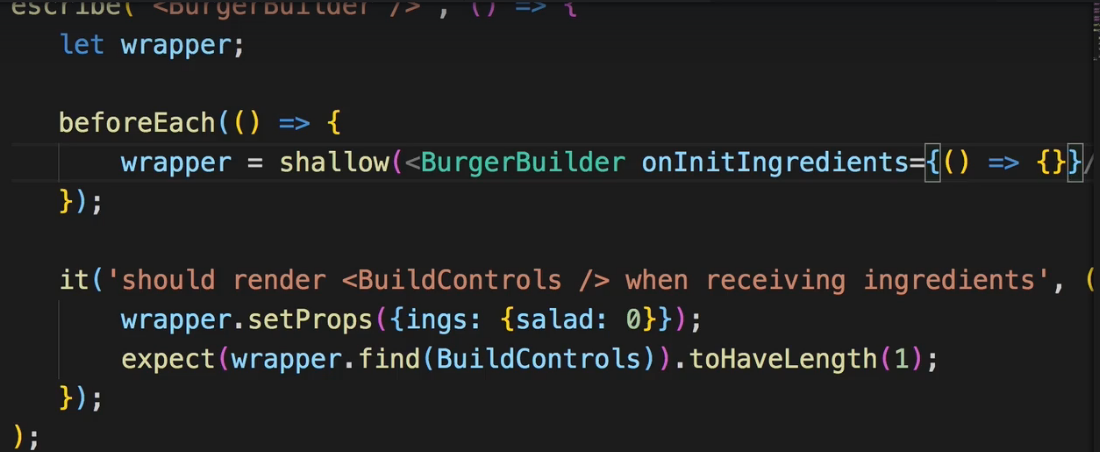
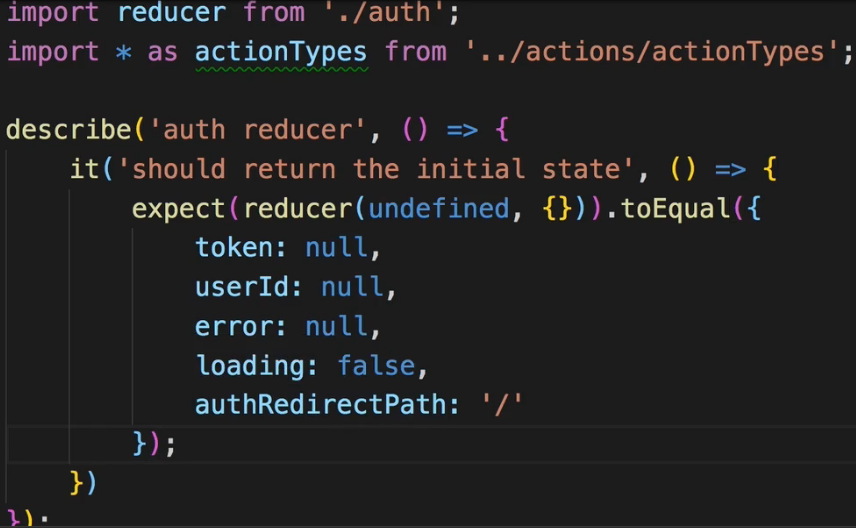
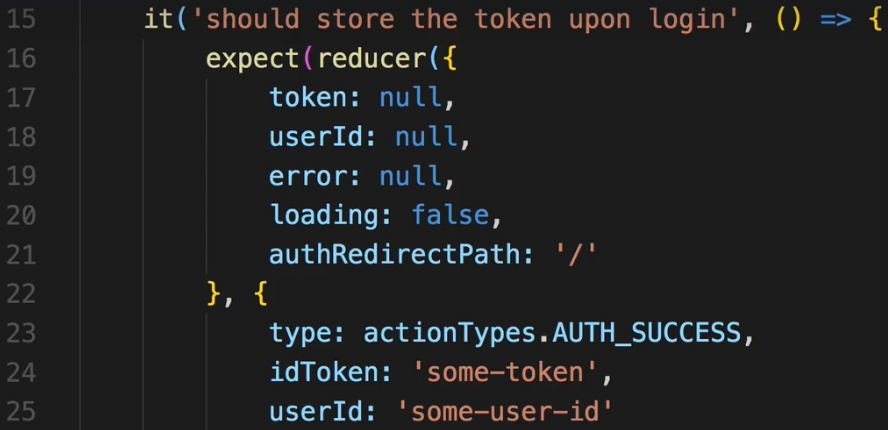
**Section 20 done: 12/12 Testing**  
**Introduction**  
\* **Testing React Apps - Because “Just Clicking Around” Doesn’t Do It**.  
\* Now with testing here I don’t mean manually clicking around in our application, we already do that constantly while developing it, with testing here I mean **writing automated tests**.  
\* And now here’s something important which I want to mention right at the start of this module: TESTING is a complex topic, and is actually huge enough topic to make up an entire course just on its own. This is not TESTING in general introduction and it’s also not a complete React testing guilde-like module. Instead, I will introduce you into writing UNIT TESTS for React applications built with create-react-app and that can then be something you build up on or which helps you apply your general JavaScript TESTING knowledge you already have or gather other resources.  
**What is Testing?**  
\* As I said TESTING doesn’t mean that we TEST our application manually - we obviously do that and we should do that.  
\* So what we typically do is: we build an application, then of course we TEST it manually in the browser and then we ship it to a server.  
\* Now there’s 1 extra step we can add in any development workflow and that is that we don’t just ship the app after TESTING it manually, but that we also have **AUTOMATED TESTS**.  
\* Now these are tests which run automatically, we write them and then each test tests a tiny fraction of our application. That’s a so called **UNIT TEST**, it tests a unit of our app. And the idea behind such tests is that since we define these tests, if we change anything in our application and that breaks our app or breaks a component in our app, then the respective test will **FAIL**. And hence we might get a warning about a potential error we introduced before we even have to find it by manually testing the browser. It’s much more important in bigger apps where it’s easy to break something without quickly noticing it.  
\* There also is a different way of thinking about TESTING - intead of adding it in the end of the build workflow, there also is this idea of **TEST DRIVEN DEVELOPMENT** which puts the tests first. There you write the tests before you write any application code and therefore all the tests will of course FAIL initially but then you add the application code and the tests should pass step by step. That’s not the concept I’ll show you here, instead, I’ll show you how to write these UNIT TESTS and you can of course then build up on this knowledge to use it in whichever way you want to use it.  
   
\* If then a TEST FAILS, we can either adjust the code or the test itself depending on the situation.  
\* **To be able to TEST, we need some TESTING TOOLS**.  
**Required Testing Tools**  
\* We need 2 tools to write good tests and be able to test our React application.  
=> 1. => **Test Runner** => this is basically a tool which is responsible for executing our tests, so for running the test code and providing a **Validation** **Library**, which in the end is a library which allows us to do comparations and potentially throw some errors.  
\* And the idea behind UNIT TESTS is that they don’t run in the browser, but instead with Node.js though often emulated to be in the browser environment with the help of specific JavaScript packages. The Test Runner then is the core tool which simply executes our code, using that environment. And the good thing is that create-react-app already comes with a pre-configured testing environment we can build up on.  
=> We will use **Jest** in this course. Jest is already installed in an app created with create-react-app and it’s a popular JavaScript testing tool which is not limited to react but often used in React apps.  
\* Now running the tests is 1 thing, when working with React and components, we also need a way of emulating these components, of basically mounting them to some non-existent DOM you could say and then traversing our components. And we want to do this in an efficient, quick way, without having to actually create that whole component tree which might also introduce some side-effects.  
=> 2. => **Testing Utilities** => for that we need Testing Utilities which help us with testing and there we specifically need the just described help to simulate the React app, mount components and dig into that DOM which is created with React.  
\* In this course, we could use **React Test Utils** - that’s the official utility tool but there’s another tool which became more and more important and interesting and is also suggested by the React team and that is **Enzyme**. Enzyme is a tool developed by Airbnb, they use React in a lot of their projects and they share this tool which makes it easy to mount components and then navigate through them.  
\* I’ll show you how to use Enzyme in this course.  
\* So these are the testing tools, that leaves us with just 1 question: What should we tetst?  
**What To Test?**  
\* Actually writing good tests is quite complex and requires a lot of practice, which is also one of the reason why this isn’t a complete testing guide.  
\* So in general here are some things you would not want to test:  
=> **Library** => you don’t want to test React or Axios itself, these are third-party libraries which already were tested by the developers, you want to test the code you add to your application, and there you want to test the code which does not use that library. So for example if you’re sending an Axios POST Request, you don’t need to test if that is sent successfully, if it fails you probably have internet connection, it’s not the Axios library. You typically want to fake data you get back from like a server in such a use case and just test what you are doing with such returned data.  
=> **Complex Connections** => Additionally, you don’t want to test too complex connections, especially in React, there is the danger of you testing that you click a button in 1 component, and you change something in a totally different component. Now you don’t have to test if React is able to use the concept of props to emit an event and pass it on, you would be interested in testing if the button click in your React app triggers a specific prop in the first place, or if the change in data you receive via props in 1 component leads to a different result being rendered, that is what you want to test.  
\* There are some testing guides which go so far to say: For React components you only need to test if a React component itself is rendered correctly.  
\* Now I will show you how to test different things with React components but definitely keep in mind: Don’t test too complex connections.  
\* So these are the things you don’t wanna test.  
\* What are you going to test then?  
=> **Isolated Units** => typical examples are isolated units, you want to test that **REDUCER function** you created, you want to test that **component function** you created, you also want to **test conditional output** - if your component has a property which leads to something being rendered if that property is true, then you want to test if this really happens. What happens if some property changes in your component. Does that affect the output correctly?  
  
\* These are the things you want to test in your UNIT TESTS.  
\* Let’s learn how we write UNIT TESTS in a React app.  
**Writing our First Test**  
\* In package.json (after ejecting) we see that Jest really is installed:  
  
\* Let’s install enzyme - that alone won’t do the trick though, we need to install 2 additional packages to make it work correctly with Jest and React.  
**npm install --save enzyme react-test-renderer enzyme-adapter-react-16**  
\* All these are now added to the package.json and of course installed in the node\_modules folder.  
\* Let’s write our first test and I want to start by writing a test for a component because obviously in React talk constantly about components.   
\* Starting with **testing a Functional component** because that of course is a super easy way of testing.  
\* I want to start with the NavigationItems component.  
\* **Now keep in mind that most of your React components are just functions and therefore they only depend on the props they receive. That’s something you have to keep in mind for testing**.  
\* Let’s create a testing file for this component next to it in the same folder: **NavigationItems.test.js**.  
\* The .test.js is important because that is automatically picked up by create-react-app once we run a special command and will then be included in the testing and therefore will be tested.  
\* A test uses Jest by default and Jest gives us a couple of methods to define the test.   
=> The first important method is the **describe()** method. You don’t need to import it in that file, it’ll automatically be made available in our create-react-app project once we run the test command.  
=> **1st argument** is just the description of the test bundle this file holds. That’s what you’ll see later in the console output. So it should be something which allows you to identify which kind of tests were run here.  
=> **2nd argument** that is your testing function. It’s a normal JavaScript function and in here you’re going to describe, you’re going to write your actual tests.  
=> You write a test by writing **it()**. That’s another function which will just be available. It allows you to write 1 individual test. It also takes 2 arguments.  
=> **1st argument** is again just a string, a description which will appear in the console. And **typically you just complete the sentence to describe what it should do**.  
  
=> So it should render 2 NavigationItem-s if we are not authenticated.  
\* It should just be something meaningful you recognize in the console because if the test fails, you want to know which test failed.   
=> **2nd argument** is again a testing function describing the actual test. Here that is just a normal JavaScript function.  
=> Now what we want to do is: we want to create an instance of this NavigationItems component as it would be rendered to the real DOM through React and then have a look into the rendered component and see what was rendered for the case that the isAuthenticated prop is false.  
\* Now you might think that for this we obviously need to render the entire React application because NavigationItems is just 1 tiny piece in the entire React application.   
=> That is where Enzyme comes in, this testing package.  
=> Enzyme allows us to just render this NavigationItems component standalone, independent of the entire other React application. That’s the whole idea behind the Enzyme package - that we can really write UNIT TESTS, isolated tests, tests where we don’t need to render the complete React app.  
\* We’ll import Enzyme but before that we need to configure Enzyme and connect it to my React version.  
=> For that I need to **import the Adapter** and it’s a default export so you may name this whatever you want, from **enzyme-adapter-react-16**.  
\* Now we’ll also need something specific **from Enzyme**, this is a named export so we need {}, the **configure** function.  
\* With these 2 imports we can now execute **configure()** and pass a JavaScript object to it, there we should set up an `**adapter:**` property and assign **new Adapter()** **as a contructor function**. So this Adapter is instantiated with new Adapter(). And that’s all, **with that, Enzyme is connected**.  
\* Now we want to render a NavigationItems component and then look into it.  
\* For this Enzyme gives us a specific HELPER METHOD we can use: **shallow** function.  
=> shallow is the best way of rendering React components in many circumstances.  
\* Enzyme offers 2 alternatives which I’ll also point you to later but shallow is the one you should use as often as possible because 1 thing shallow does is: it renders the component with all its content but the content isn’t deeply rendered, so the NavigationItems component here has NavigationItem components but these are only rendered as placeholders, the content of them isn’t rendered. And that of course again is important for creating isolated tests when we don’t then render a whole sub-tree of components, we just want to render this component and know what’s inside of it without rendering everything which is nested inside its included components.  
\* The const name is up to you but `wrapper` is often used.  
\* **You pass JSX to the shallow() method**.  
\* We now write our expectation.  
\* For that we use another method which is made globally available by Jest: **expect()**.  
=> Inside of expect() we define the thing we want to check. So here I want to check if the wrapper contains certain element. On that wrapper we can use a UTILITY FUNCTION probided by Enzyme:  
**.find()**.  
=> This allows us to look into the wrapper and see if it contains a certain content. And here I want to find a NavigationItem.  
\* Important: **this now is not a JSX element**, it’s this normal exported function from the NavigationItem file.  
\* You could say the expectation is that we do find it but tests are a bit more flexible than this. You could expect that you find it once or twice or that you don’t find it - maybe you want to test for the opposite.  
=> So here we have to chain a UTILITY METHOD made available by Jest: **.toHaveLength()**.  
\* It will have isAuthenticated set to false because we’re not passing this prop.  
  
  
\* So with that, this test should actually succeed.  
  
\* We need the “test” scrypt to run our tests.  
**npm run test  
npm test**  
\* Like start, it’s a special scrypt where we don’t need the run command.  
\* This will execute all the tests and will automatically scan for all test files by looking at the ending: test.js.   
!!! In case you get a blocking error at this point, delete the App.test.js file and run it again.  
\* This warning regarding the polyfills can be ignored.  
\* **Test Suites**: that’s the decribe() functions.  
\* **Tests**: that’s the it() functions.  
**Testing Components Continued**  
\* Let’s now write another test, for the same component.  
\* Well of course we could test the opposite: we want to have 3 NavigationItem-s if we are authenticated.  
  
\* Now of course we can write multiple tests like this but if we constantly do the same at the beginning of the test, namely we render the wrapper even though the props changed but we can do this differently too, so if we constantly do the same, there’s a HELPER METHOD we can use inside the describe() function here. It’s the: **beforeEach()** function. As the name suggests, this is a function which **will automatically be executed before each of your tests**. You also have a **afterEach()** function for cleanup after all your tests if you need to.  
=> And in the beforeEach() we can do some general setup of course.  
=> It takes a function as an ARGUMENT and this is the function which gets executed before each test.  
\* Now with that we can define the wrapper and simply use it inside it() functions, however, if we want to pass a prop like isAuthenticated, we can either assign it again inside like this:  
  
or we can use another helpful method of the Enzyme package.  
=> We can access the wrapper and then there is a **setProps()** method we can execute on that wrapper. So on anything which stores shallowy or other rendered React element.  
  
\* This is how you can write multiple tests and have them work together nicely.  
**Jest and Enzyme Documentations**  
\* For testing I can only recommend looking into the official docs as well as - if you really want to dive deeper into testing - checking out some dedicated resources, some dedicated courses or tutorial articles I mean with that.  
**Enzyme API**: <http://airbnb.io/enzyme/docs/api/>   
**Jest Docs**: <https://facebook.github.io/jest/>   
\* Always keep in mind Jest is not bound to React or create-react-app, it’s a JavaScript test runner, you can use it in any JavaScript project.  
\* In Jest docs you can find some general idea about how it works and how you configure it.  
\* And then the API Reference is especially useful, if you click on the Expect method. There you learn how to actually use the expect() method and which functions to chain to it. And a similar documentation is available for all the UTILITY METHODS here.  
\* In API Reference > Mock Functions - how to mock certain things, mock means: REPLACE. That’s especially useful when working with some Async code where you don’t want to execute that actual code, you don’t want to reach out to the web, you want to mock this, you want to step in and replace that function with a function that just returns some demo data you could work with because you’re always not testing a service - you’re testing what your code does with the returned data.  
\* Now there’s a similar documentation for Enzyme.  
\* There are GUIDES on how to use it with various test runnes such as Jest.  
\* In the API Reference you can find the shallow rendering and 2 alternatives.  
\* Full Rendering is useful if you want to render the complete component tree, that should rarely be the case but you might have some circumstances where you want to test some cross-component dependency. Always see if you can solve it with a UNIT TEST though.  
\* If you click the Shallow Rendering, then on the left you can find all the HELPER METHODS you have.  
\* Now let’s use the documentation and check the **contains()** method because I want to write one other before we leave our NavigationItems component. There we already test for a lot of things regarding the NavigationItem-s.  
=> Now I want to test something very specific. If we have a look at the NavigationItems.js file, we notice that there is a “/logout” component which is only there if we are authenticated which makes sense. So I want to check if that specific NavigationItem is there. And for that I can use this contains() helper because the contains() helper - unlike find() - does not take an element type or a CSS selector to be precise (for find() you can also use class selectors) but contains() takes a real node and you can therefore check if you have an exact match. Since we don’t use a CSS Selector, we use JSX again. And the NavigationItem I’m looking for is actually not a self-closing one but a one which has an opening and closing selector because inbetween there will be a “Logout” text. And it will have a prop `link` which leads to “/logout”.  
\* So I’m basically writing the same code as in my NavigationItem component here:  
  
=> Now I could use toHaveLength(1) or to mix it up, I can use **toEqual(true)**.  
  
**Testing Components Correctly**  
\* We might also want to test the class we get in NavigationItems or something like that but in general this looks like a good test coverage to me.  
\* Writing good tests is complicated.  
\* It’s easy to write 100 tests for a given component and test all kinds of things while missing the 1 important thing you should’ve tested. And on the other hand, you might only need 1 test to really verify if a component behaves the way you want it to.  
\* You can just practice, test for different things no matter if they make sense to be tested or not. Learn how to test and think in test environments. And learn how to use the different functions provided by Enzyme and Jest. It really is all about practicing and testing takes experience. The best way to start with testing is to always have a look at your component or your function or whatever you are testing and see: what are the crucial things that change depending on some external influences - like here the isAuthenticated prop which changes what gets rendered and then this should be what you write the tests for, so that whenever you change something in that NavigationItems component and you accidentally mess up like missing a ! before a condition or something like that, now we get a failed test because this also re-executes tests if you change your normal code.  
\* So identify crucial parts just like that - what can I easily break and what would then totally break my application behavior and then you write tests for exactly that.  
**Testing Containers**  
\* The tricky part about containers is that they are connected to the Redux STORE. The Redux STORE is some external influence on this container, if it weren’t connected, testing it would just be very equal to the other components because then yes it might have state but Enzyme also has methods to handle this - just as we have setProps(), we also have **setState()** to simulate different states in a component.  
=> So the tricky thing really is the Redux STORE.  
\* Now the good thing is: we don’t really need to test the connection of this container to the Redux STORE, we can rely on the Redux STORE to work correctly. And then in the end we only receive data from the STORE as props to this container. So we’re back to the previous world - we can just simulate props in our tests because we want to simulate different outcomes in different states of props anyways, so we don’t want to connect it to some real STORE, we don’t want to if the STORE passes this correctly to this component, that’s the job of the connect() method and that’s coming from a third-party library, we can rely on that to work correctly.  
=> So what we really need to do is: we need to get access to the component behind this container so to say.   
=> And 1 convenient trick is to simple EXPORT this BurgerBuilder class.  
  
=> So simply add the EXPORT statement in front of the container.  
=> This now gives us a named export in this file - BurgerBuilder which gets access to this class which is just a React component - and we still have the default export we use in our app.  
\* Let’s create BurgerBuilder.test.js.  
\* This can now be rendered again with shallow() and so on just as we rendered the normal component because it is just a normal component and we totally strip out the connection to Redux which is what we want.  
\* Let’s quickly have a look at what we could test here: it would be interesting to see if we actually have BuildControls here if the `this.props.ings` are non-existent because we should only render that if he ingredients are passed along.  
  
  
  
=> I can use find() to find my BuildControls - not a JSX element, just the type here and this should have a length of 1.  
!!! In case you’re ever getting this error regarding App.test.js, remove the App.test.js file with which the project start because it will actually mount your entire application and fail due to using Routing.  
=> Now you should get this error:  
  
=> This error makes sense because shallow() will render your component here even though it doesn’t deeply do that and in the BurgerBuilder you see that in componentDidMount() we call `this.props.onInitIngredients`. And this of course is not made available because we only set 1 prop, the `ings`.  
=> So to fix this we have to add this as a prop.  
=> Now setProps() won’t do the trick because that is set after the component has been instantiated so that’s too late, we have to add it here:  
  
and I’ll set it to an empty arrow function to simply fulfill the requirement of passing a function.  
\* Now the test passes, if we change `ings` to null, it fails.  
\* And this is an example which you of course may fine-tune to your need, to see how you can test containers - you essentially strip out the component part and get rid of the connection to Redux.  
\* How do we test Redux then?  
**How to Test Redux**  
\* So we learned how to test containers and I mentioned that the important part is that you don’t test the connection to Redux.  
\* Do we test Redux at all?  
=> Yes, but we have to be careful about what we test.  
=> We probably don’t want to test very complex chains of ACTIONs and REDUCERs and STATE.  
=> In the end the REDUCERs are the meat we wanna test, especially if we follow the pattern of not putting too much logic in the ACTION CREATORs.  
=> Then testing REDUCERs is super simple, they’re Synchronous so we don’t have to deal with Async code and they’re just functions - we pass something in, we get something out.  
\* Let’s add an auth.test.js file.  
=> Here we don’t even need Enzyme because we’re not testing any React components, we don’t need to render anything, we just test normal JavaScript code, we test functions, the REDUCER function.  
=> So we import the REDUCER and the ACTION TYPES.  
\* Obviously we can again use beforeEach() to do some general configuration if we need to but I want you to start right with the tests.  
\* So what do we test?  
=> Let’s for example test that we get the right initial state if we actually pass an invalid ACTION TYPE to it. And I now expect() my REDUCER - if I execute it - so inside expect() I simply execute the code I want to test, just as before with wrapper but now with a normal JavaScript function. And I’ll actually pass undefined here as the initial state - that is the case when the state is just getting set up at the beginning of our app and the ACTION is just an empty object so no specific ACTION. And it should equal the initialState so let’s copy that.  
  
\* And this is how easily you can test a REDUCER.  
\* Of course you can also test for specific cases.  
\* Now let’s execute the REDUCER with the initial state and with an ACTION and its PAYLOAD.  
  
  
\* Now we expect it to spit out an updated state which respects this token and user ID.  
\* Now it passes.  
\* And as soon as I change the expect userId to “some-user”, it fails because now they’re different.  
\* So this is how we can also test REDUCERs, they’re **PURE FUNCTIONS** and therefore very simple to test.  
**Wrap Up**  
\* One of the most important things when it comes to testing is the way you think about tests.  
\* Don’t test unnecessary stuff because that unnecessary stuff often is the most difficult to test.  
\* If you’re testing the whole workflow from: I click a button to: my state updates to: then something changes in another component, then you won’t have fun simulating that, that would require a lot of difficult work and code where you need to MOCK the HTTP calls so that you don’t actually reach out to the server during your tests and all of that isn’t fun at all and it’s not only no fun but it’s also redundant, because we don’t need to test if we can reach out to the server, we need to test:   
=> Does our REDUCER work correctly?  
=> Do we update our components correctly if the input changes?  
=> Maybe we want to test: Do we fire the correct prop if we click a certain button?  
\* All these things can be easily tested in isolated UNIT TESTS and this really is the way to think about it and the way I encourage you to practice.