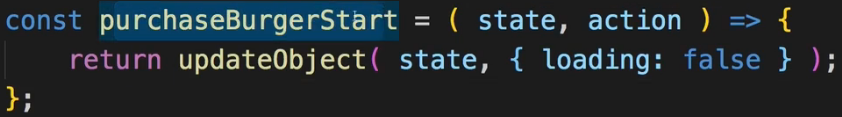
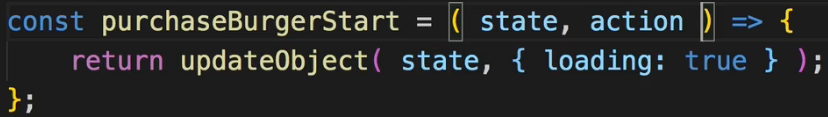
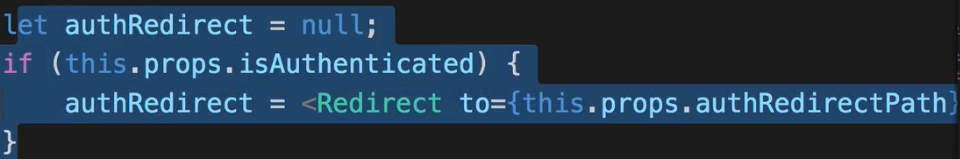
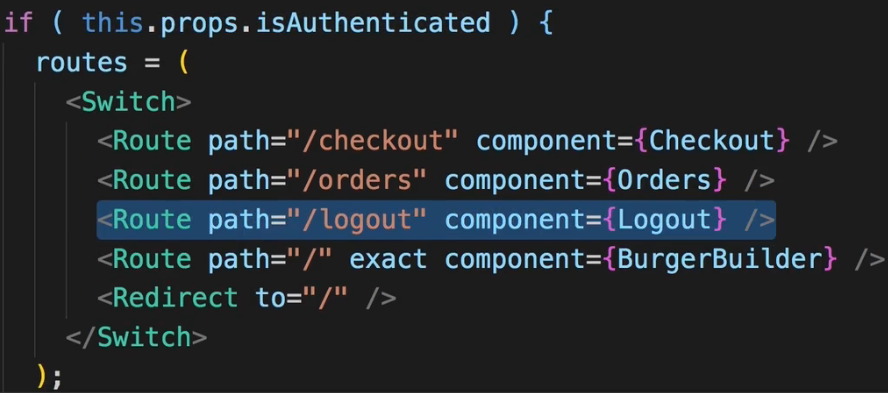
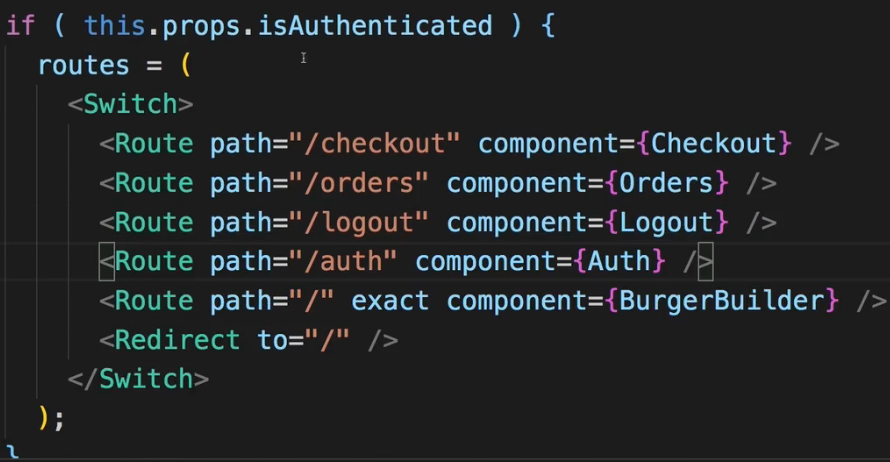
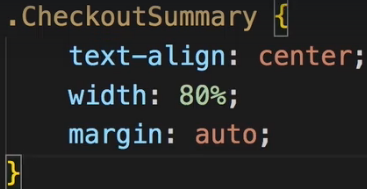
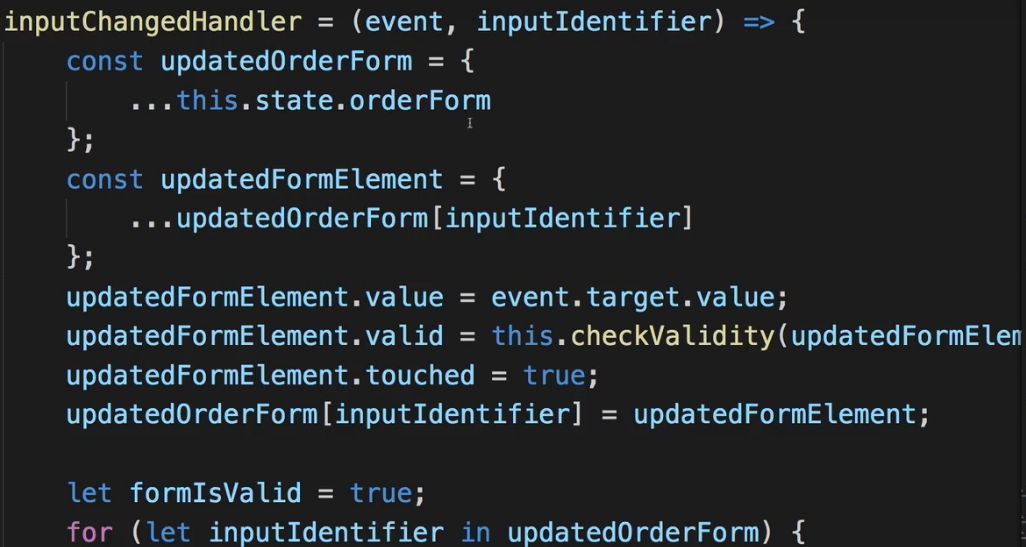
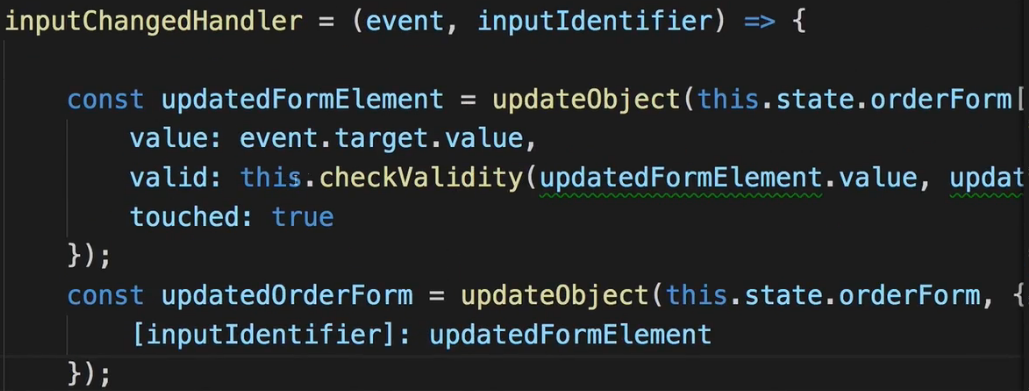
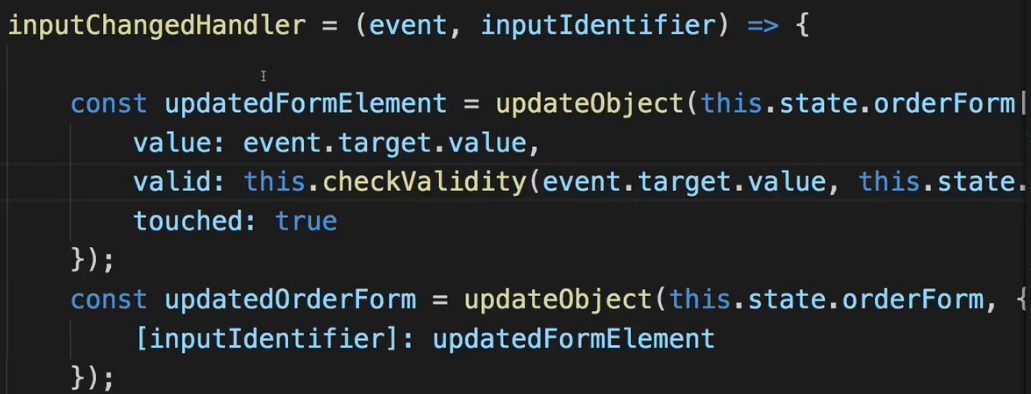
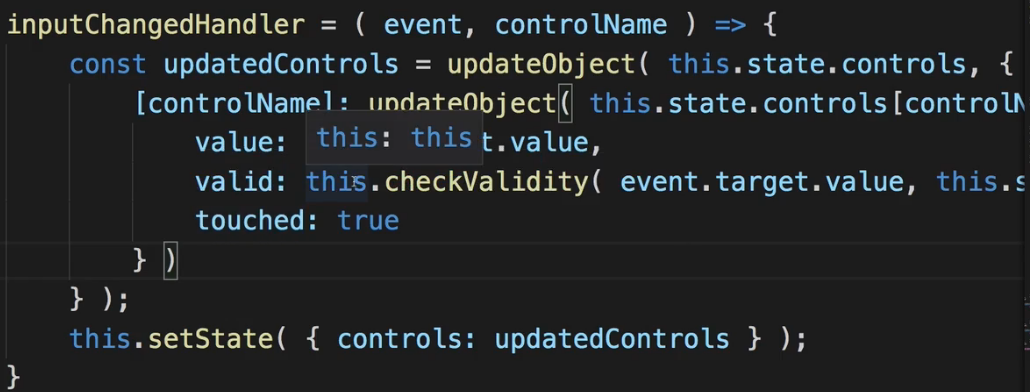
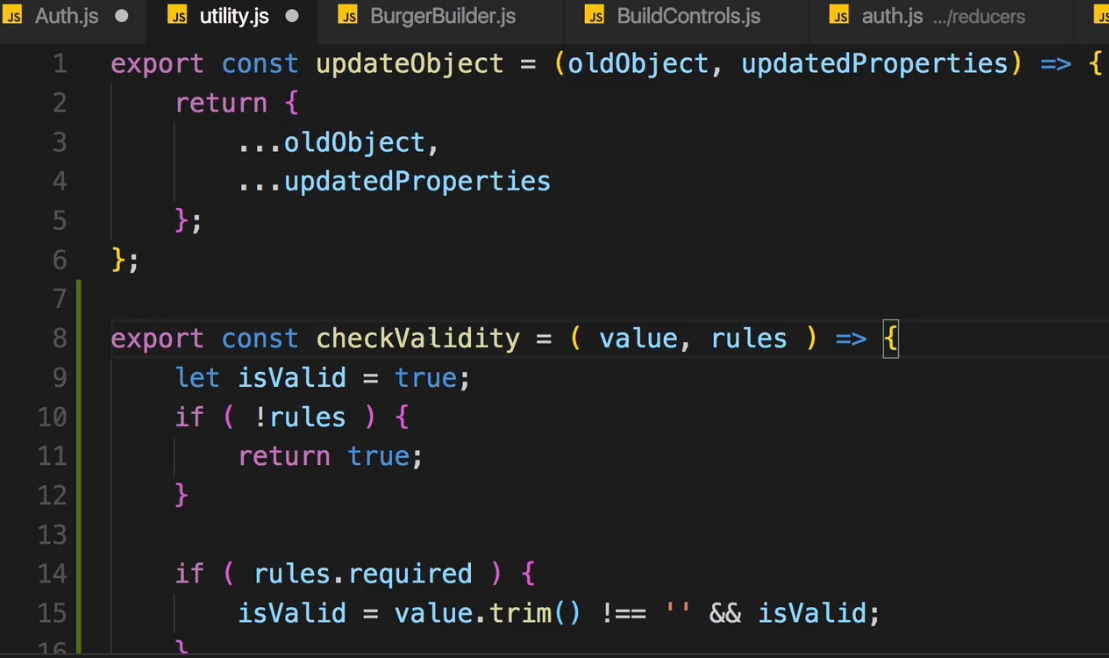
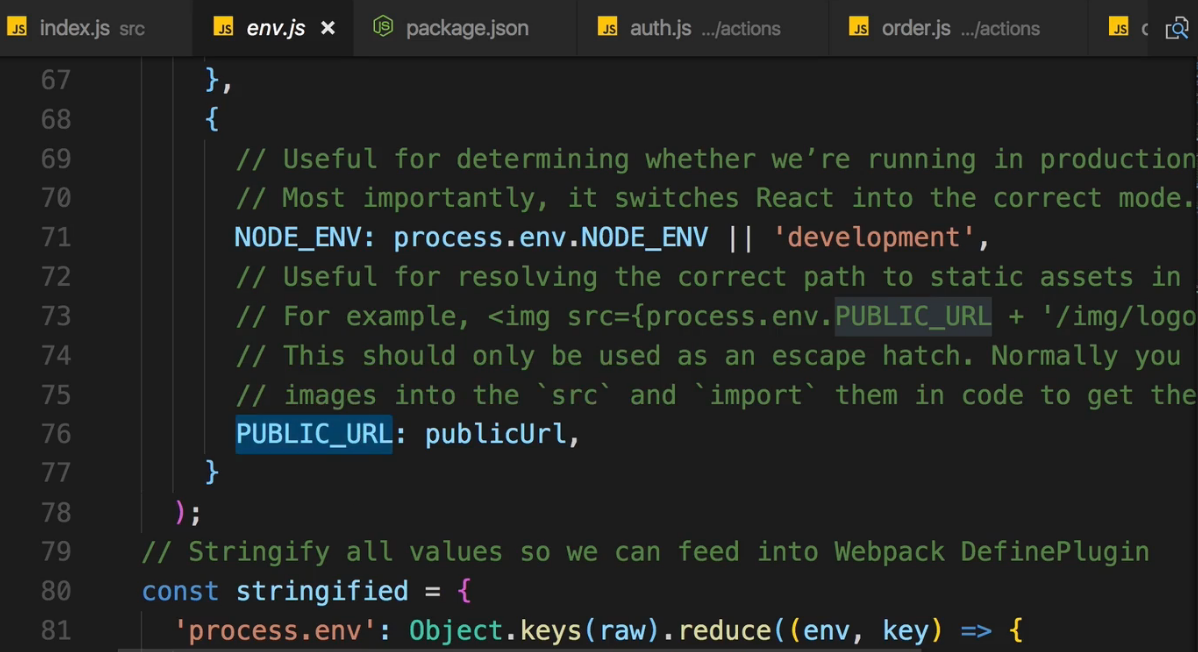
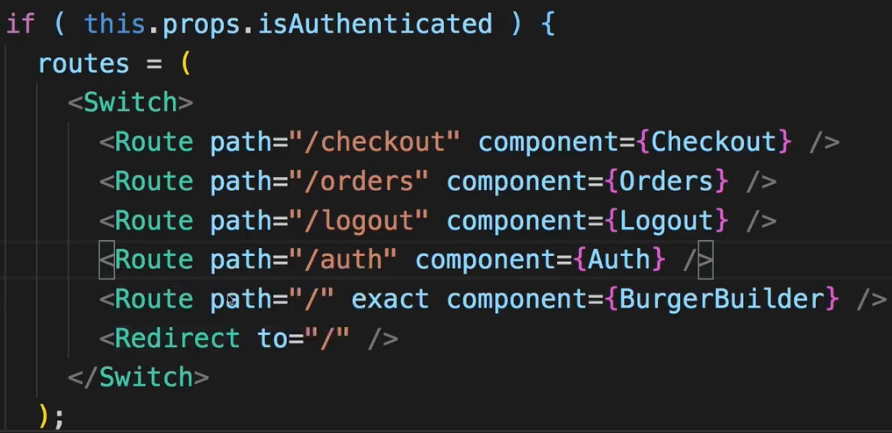
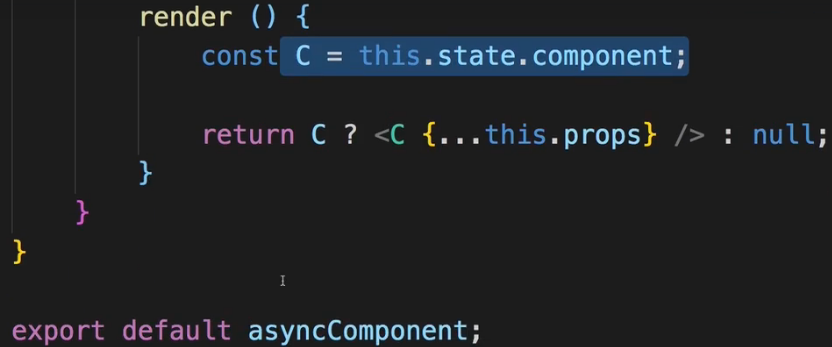
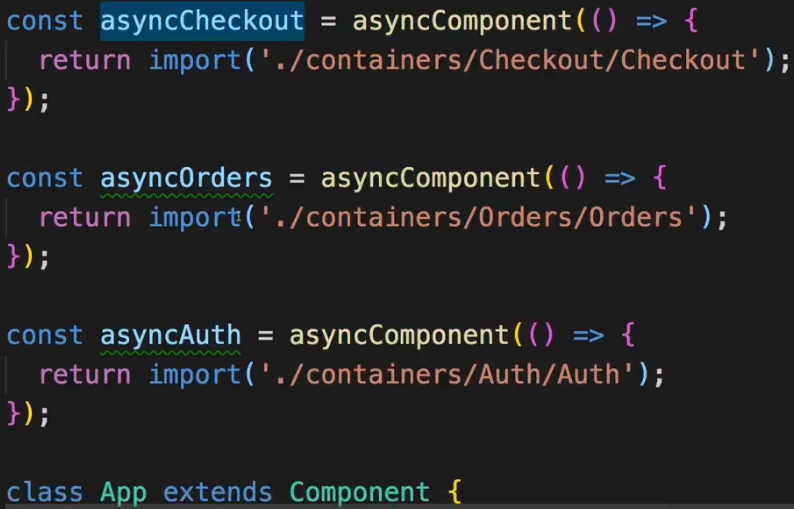
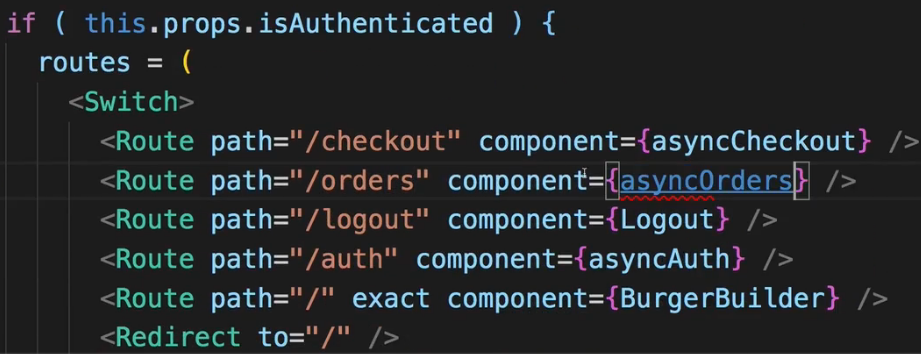
**Section 19 done: 9/9 Improving our Burger Project**  
**Introduction**  
\* Improvements & Fixes: Polishing the Project.  
\* I just want to walk over our Burger Builder application and analyze it, fix some tiny glitches it has, and see what we can improve about it.  
\* I want to have a polished project after this module which works and feels nice.  
\* It already does so but there are some things we can improve.  
**Fixing the Redirect to the Frontpage**  
\* There is a tiny issue when we place a new order.  
\* In order.js REDUCER:  
  
=> **The purchaseBurgerStart which of course is dispatched when we click on “ORDER”, when we send the order to the server, we set loading: false. Now that doesn’t make any sense at all because here we want to set loading: true and we had it that way, we accidentally set it to false when refactoring the REDUCER. That way we show the Spinner**.  
  
\* Now there’s another improvement.  
\* **Let’s build a burger, log in, and we are redirected here back to our front page**.   
\* Now that shouldn’t happen, we fixed this in the authentication section that we should be redirected to checkout and we were for a fraction of a second but somehow it then decided to redirect us back to the front page.  
\* So here we did correctly adjust setAuthRedirect but still it decided to redirect us to the front page.  
=> The reason for this problem with this wrong redirection is that we do redirect in the Auth.js container:  
  
=> Here we use our redirect path to redirect the user.  
=> The problem is that this happens in the Auth.js container. And in our App.js container - in our root app component where we render different routes for different authentication statuses, there when we are authenticated, we don’t render the Auth.js component hence the code in the Auth.js component which would redirect us correctly to the checkout page, will never be used.  
   
=> To fix this we can reintroduce the Auth.js route.  
\* Now of course the disadvantage is that now authenticated users can visit the auth page, on the other hand they can’t do any harm in there.  
\* You might of course be able to find other solutions to redirect anywhere else but this is a quick and easy and safe solution to get back to the original behavior.  
\* **Now another issue we face is if I build a burger and I am logged in here, in that burger builder here you will see that on certain screen sizes the burger is positioned incorrectly, it’s not centered**.  
=> The reason for this is that the burger has a width which is adjusted to the viewport but it’s always inside a DIV which takes 100% of the width of its wrapping element and that would be a DIV which also dictates a certain with (80%) - the CheckoutSummary. So the CheckoutSummary is dictating a certain with for our burger and that is influencing how our burger is displayed here.  
=> To fix that we can go to CheckoutSummary.css and there where we set the width to 500px for wider displays, I don’t want to do that, I want to stick to the 80% I have by default.  
  
\* Now the burger is nicely centered on all device screens.  
\* **Now there’s 1 other visual glitch I want to get rid of and that is on small devices. If we open our side-nav and we click on a link, it navigates there correctly and without errors, but it doesn’t close the side-nav, I have to manually do that. Of course it should close the side-nav whenever we click a link here**.  
=> To fix this, we have to go into our SideDrawer and we want to make sure that when we click a NavigationItem, we close that SideDrawer, so we essentially want to execute this `closed` method - the method we get on the `closed` prop in this SideDrawer component.  
=> To do that we could for example simply listen to a click anywhere on this SideDrawer, that of course would mean that not only clicking on links would close it.  
=> We could also go to the NavigationItems and try to register onClick listener there too.  
\* I’ll go with the general closing strategy here.  
  
\* Now if I click anywhere in the SideDrawer, it closes, but that should be alright. If I click on a link, it therefore also closes.  
**Using updateObject in the Entire App**  
\* There are other things we can optimize in our application.  
=> **For example we do use this updateObject() UTILITY in our REDUCERs, there we call updateObject() all the time. Now if we have a look at our ContactData.js component, there we will see that in the inputChangedHandler() we’re also kind of updating objects - the orderForm and the orderFormElements in there**.  
\* Right now the UTILITY FUNCTION is in `store`.  
=> I’ll create a folder `shared` in the root part and there I’ll move my utility.js file.  
\* We need to adjust our imports now in all our REDUCERs.  
  
\* Now we can access it from anywhere in our application without our path looking strange because why would a general UTILITY be placed in the `store` folder.  
\* In ContactData.js:  
  
[inputIdentifier], {} props to overwrite  
  
=> Now we’re updating the form element by taking the old one and updating it safely with the updateObject() HELPER METHOD.  
\* The inputIdentifier is the name, email, street, … .  
\* Now 1 note - checkValidity here where we use `updatedFormElement.value` that now has to be `event.target.value` and `updatedFormElement.validation` should be `this.state.orderForm[inputIdentifier].validation`.  
  
\* If we check it, it’s all working.  
\* Now we’re not limitted to this file, the Auth.js container is similar - there we also handle a form and there we also update our controls.  
  
  
\* So again I’m using updateObject() to have a little bit leaner code and less duplication.  
\* You can of course use it in other places or create more UTILITY FUNCTIONS when they make sense.  
**Sharing the Validation Method**  
\* **There’s 1 other feature we also share across multiple files and hence it would make sense to outsource it**.  
=> **The validation logic - checkValidity(). It’s the same method in the Auth.js container and the ContactData.js container**.  
=> So let’s grab it and put it into our `shared` folder utility.js file for example. You could also create a new validation.js file there.  
  
\* Of course it’s no longer a method, here it’s a function, so let’s export it as an arrow function. Inside the function nothing has to change because we’re only working with the function arguments - as we should in a good function.  
  
=> And replace this.checkValidity() with just checkValidity(). And that’s all and this makes this file a little bit leaner and removes this unnecessary code duplication. Let’s do the same for ContactData.js.  
**Using Environment Variables**  
\* **There’s 1 other thing I want to bring to your attention - if this project gets deployed, everyone will be able to look into your STORE with the Redux Devtools. That’s not necessarily a problem, in the end people can always look into your JavaScript code if they really want to, you can’t encrypt it. But you maybe want to make it a bit harder and not as obvious to access your state**.  
=> To do that, you can go into your main index.js file which is where we unlock the Redux Devtools:  
  
and we can adjust this code a tiny bit to only unlock it if we are in DEVELOPMENT MODE and not do it if we are in PRODUCTION MODE.  
=> To do that, we can take advantage of a feature called **ENVIRONMENT** **VARIABLES**.  
=> In our create-react-app project, and this is a feature which is project-setup-specific, we can use ENVIRONMENT VARIABLES, we can define them in the `config` folder. There at the end of env.js file, you can basically add your own ENVIRONMENT VARIABLES. You can add them down here where you see NODE\_ENV and PUBLIC\_URL.  
  
\* Now the NODE\_ENV thing here is interesting - it’s automatically set for you and it is ‘development’ if you’re in DEVELOPMENT MODE.  
=> So with that we can go to our main index.js file and just need to access this variable to only set this Redux Devtool extension if we are in DEVELOPMENT MODE. We access our ENVIRONMENT VARIABLES and again this works due to this specific project setup.   
=> We call `**process**` which is a GLOBAL VARIABLE, we don’t have to import it.   
=> There we access `process.env.NODE\_ENV`  
  
  
=> So if the first part is null, we’ll use the default `compose` function.  
**Removing console.log()s**  
\* **In Checkout.js file there’s a warning about `actions` import which we never use. Let’s delete it**.  
=> That’s in general an important thing - have a look at your console output, it shouldn’t show any warnings, remove unnecessary imports to prevent bloating your final bundle size.  
\* It is also a good idea to clean up your application, not because these console.log() statements would be that bad but they will be there in the PRODUCTION too and they will give insights into your application and besides that you don’t really need to spam the console with output that doesn’t matter to you anymore.  
\* In the console we can see the files and lines where the console.log()s happen. Let’s remove them.  
\* After removing the console.log() in componentWillUpdate() in OrderSummary we could turn this again into a Functional component because all we do is just render something. I’ll leave it as Class-based one but you could do it to optimize this even further.  
**Adding Lazy Loading**  
\* There’s 1 more optimization we can definitely introduce - to optimize the way we load our routes.  
=> We can use **Lazy Loading**.  
\* Let’s have a look at the App.js file.  
  
=> We have the “/checkout” and “/orders” route and both are not necessarily visited by us, we might never go there. Now the “/auth” part maybe also never gets visited.  
=> **We can load them all Asynchronously, Lazily**.  
=> I’ll first of all add my **HOC** - **asyncComponent** to the `hoc` folder, now paste in the code I wrote in the Routing section.  
  
  
=> So this asyncComponent function which takes a function as an input which it then executes down here. This `importComponent` function will use this dynamic import syntax and then give us a PROMISE where we eventually get the component we want to load and where we then render() this component.  
\* Let’s now use it in the App.js file to load some components lazily.  
  
\* We pass a function as an argument to the asyncComponent() which will eventually return this import() statement as a function where we can then define the path to the component we want to load lazily.  
  
\* Now we got all the async components, let’s now use them.  
  
\* And now all these async components are loaded only when needed.  
\* In the Network tab, we can see any network Requests and we should see 1 I click on Authenticate.  
\* Indeed, we load `2.chunk.js`.  
\* With that we improved our application because we now only load the code we need to load.  
\* **That is a great improvement and 1 important step before we actually build our application for PRODUCTION, though as I also mentioned in the Routing module, Lazy Loading is not always better. If the lazily loaded modules are very small, as they are in our app to be honest, you might not really gain anything from adding Lazy Loading**.  
\* Still I want to show it because it is an important concept and something you should consider in optimizing any application you build.  
**Wrap Up**  
\* With that we finished the optimization of our project.  
\* Feel free to do more, to organize your folders differently or outsource more functions into HELPER FUNCTIONS.  
\* I think we have a nicely optimized, well working application which looks good, which behaves exactly as it should behave, also on mobile devices and which allows us to really present a pleasant User Experience to the user and use a lot of the features React gives us.  
\* Now in the next modules, we’ll fine-tune this application to be ready for DEPLOYMENT - with that I actually mostly mean that **I’ll show you how you could TEST the application automatically**, then we will DEPLOY it and then we’ll have a totally finished React app.