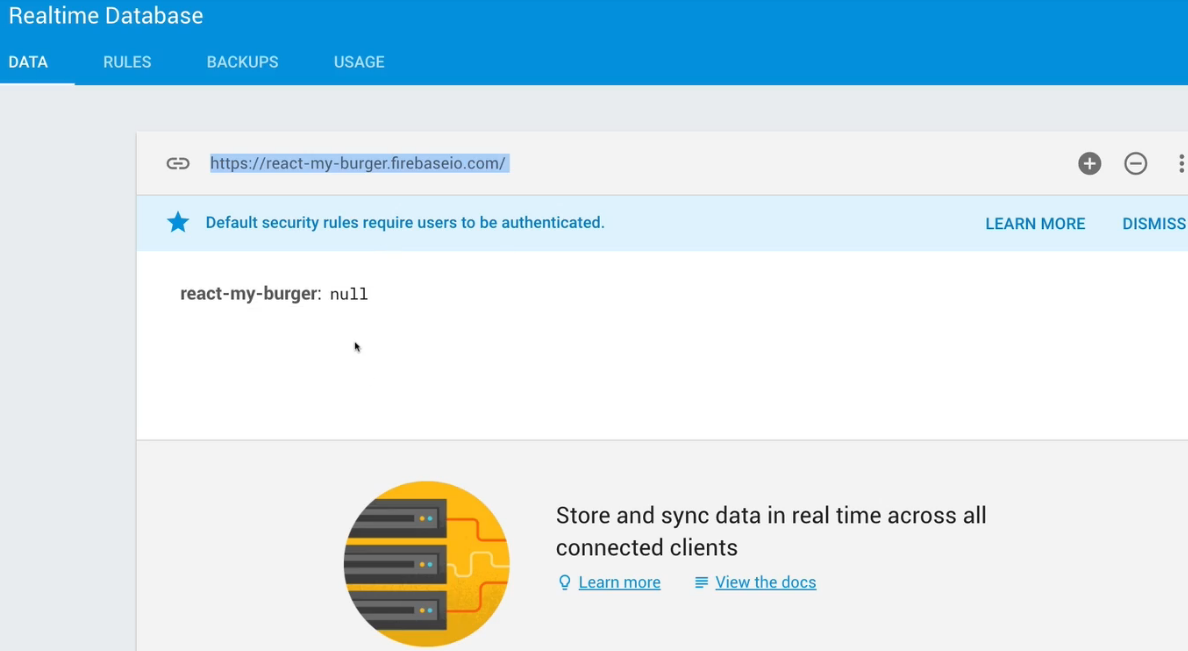
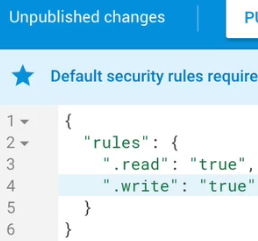
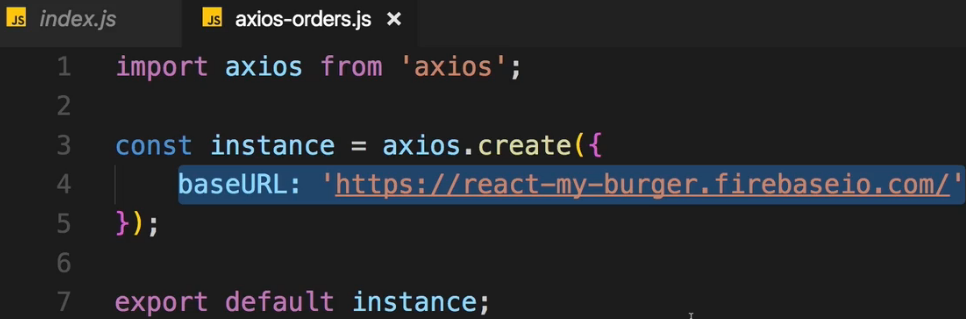
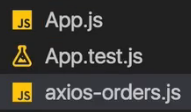
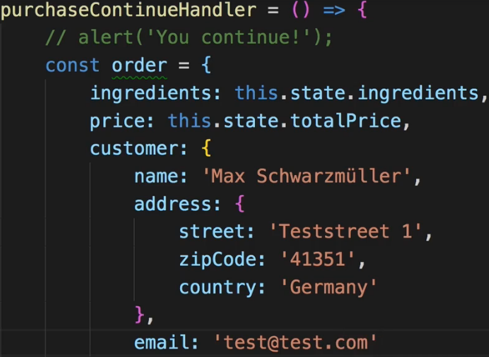
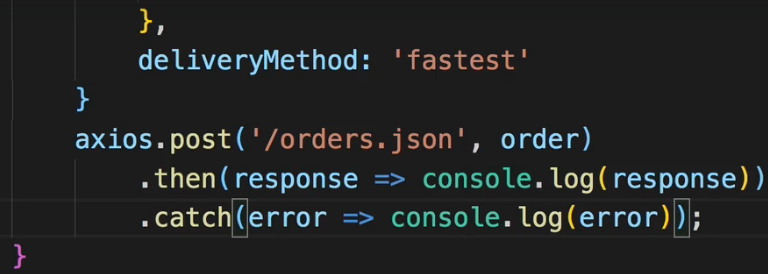
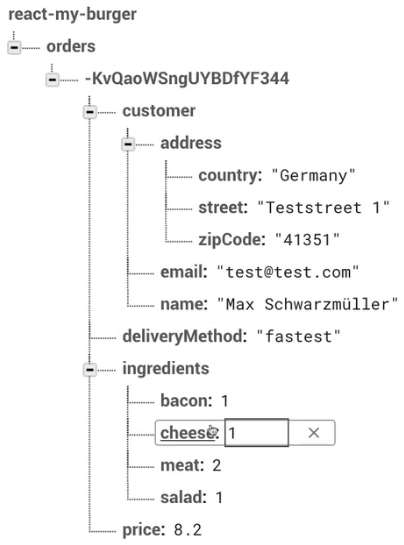
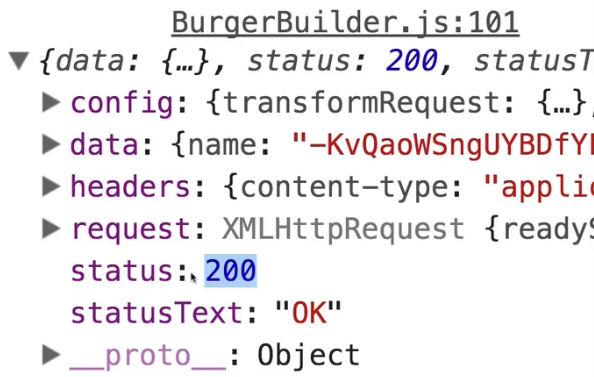
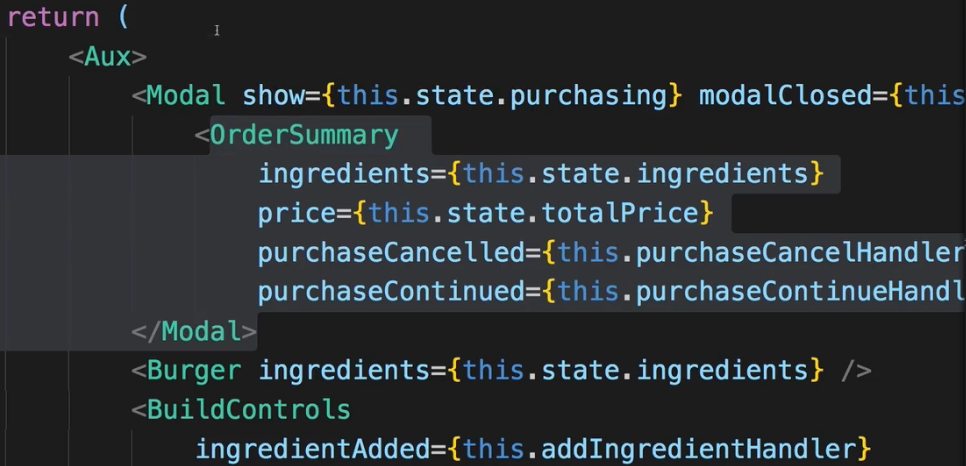
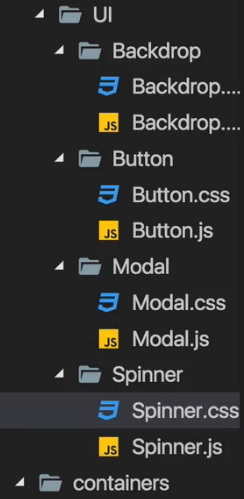
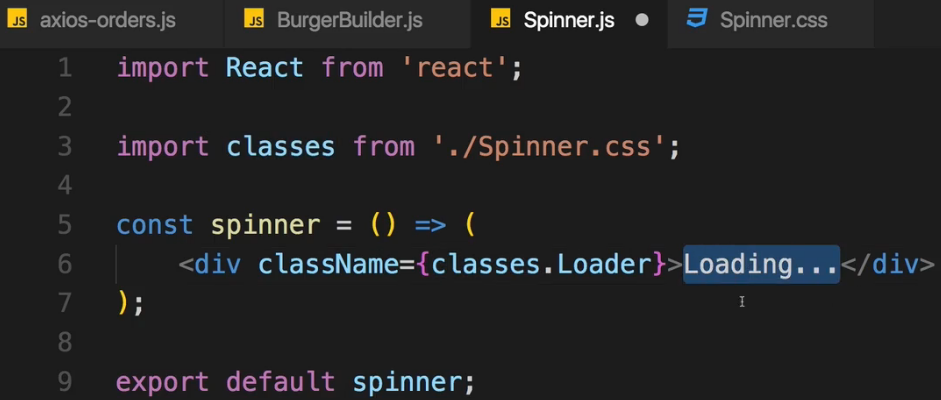
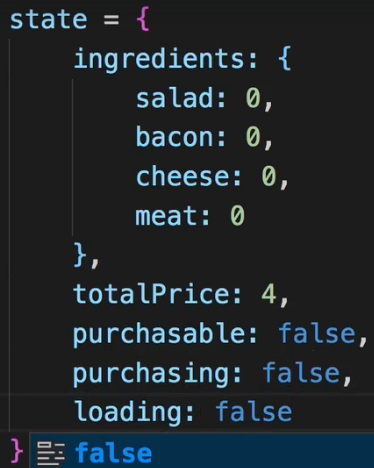
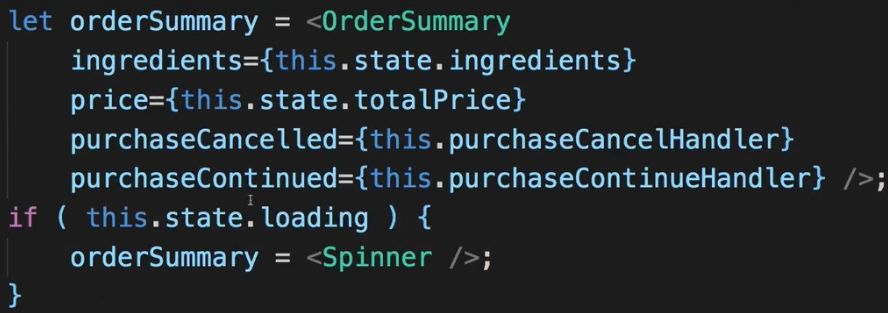
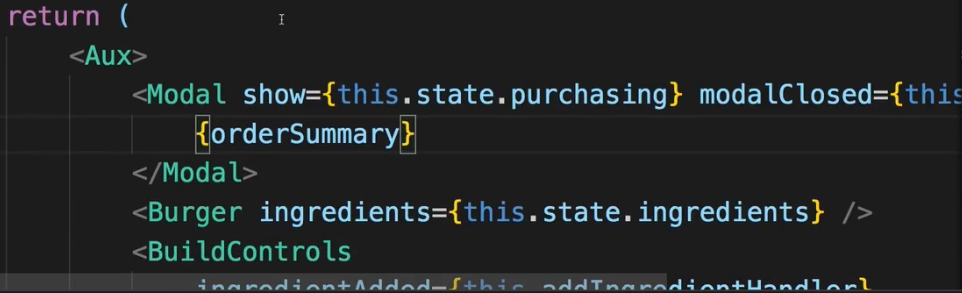
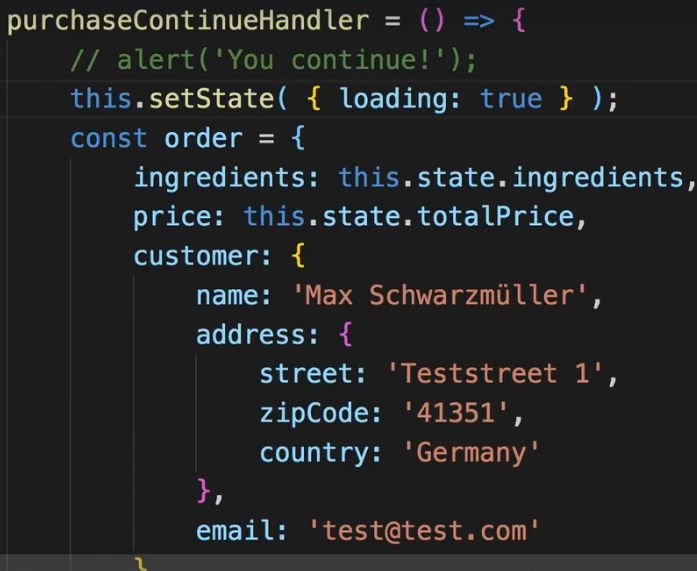
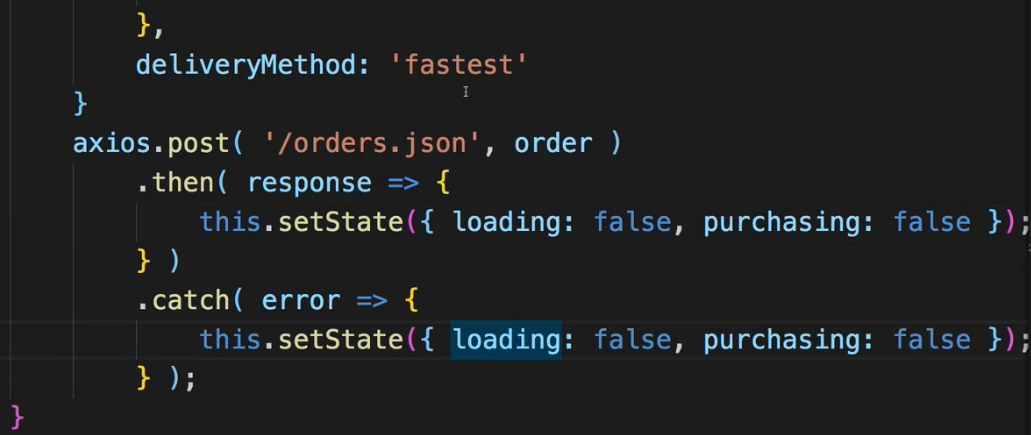
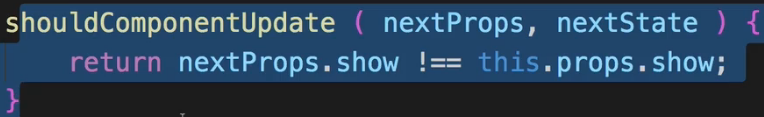
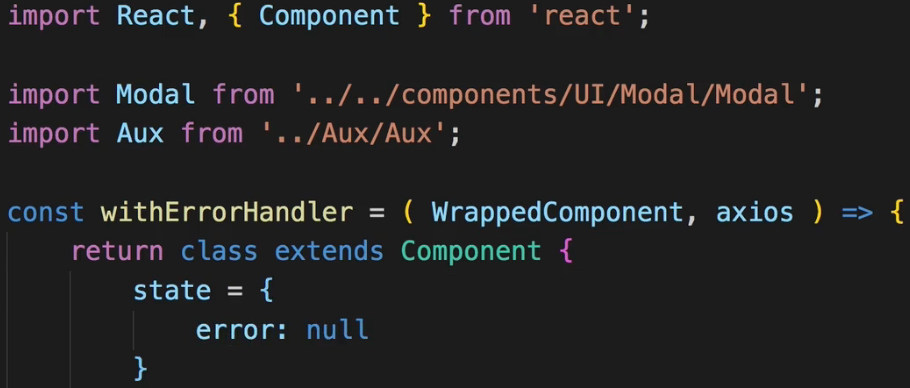
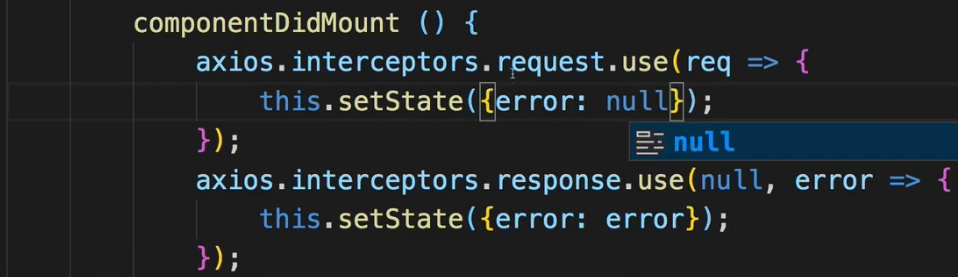
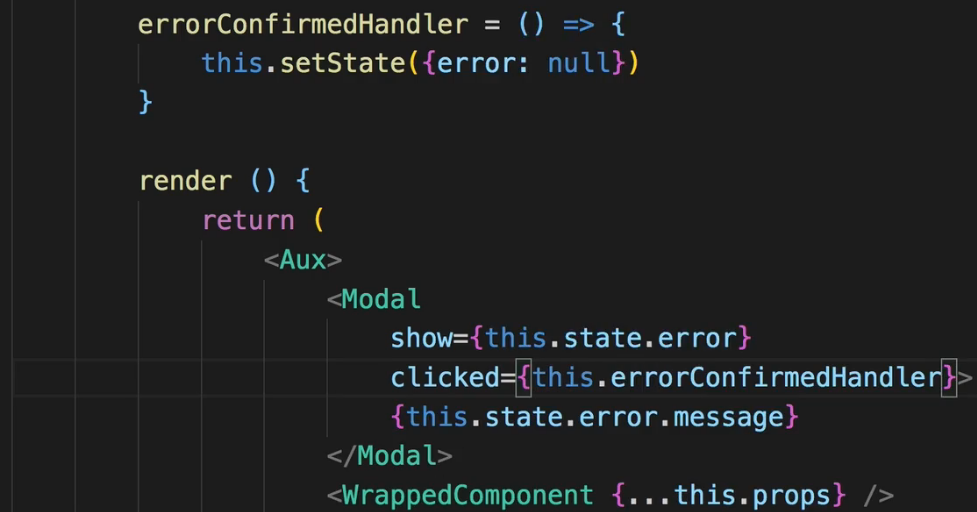
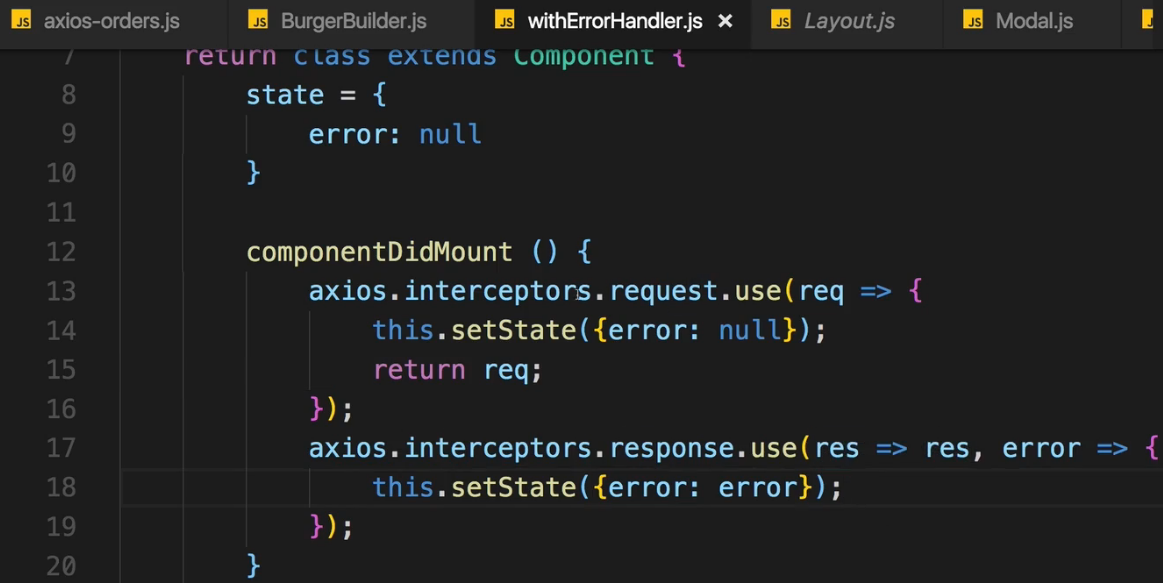
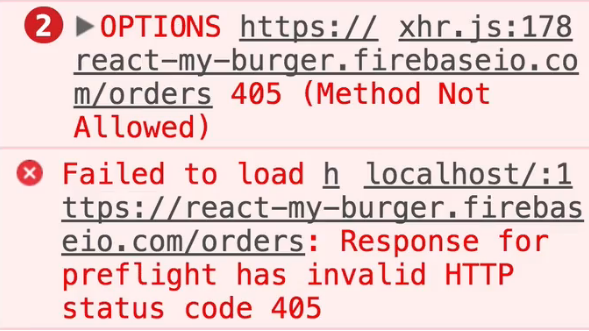
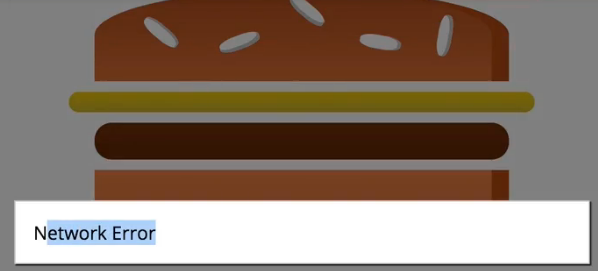
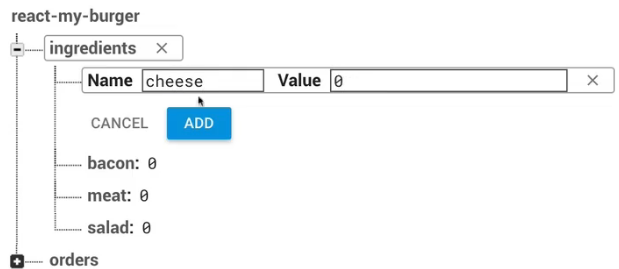
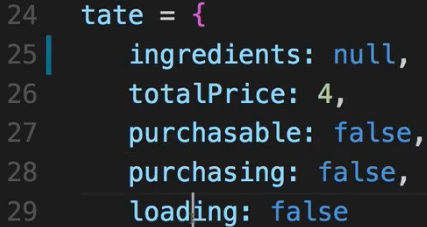
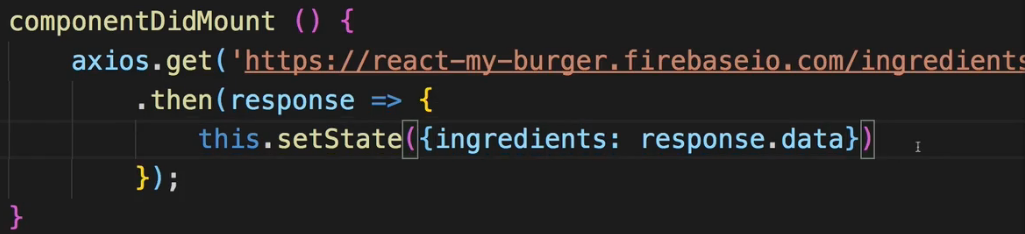
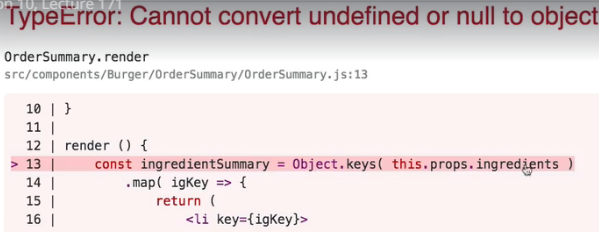
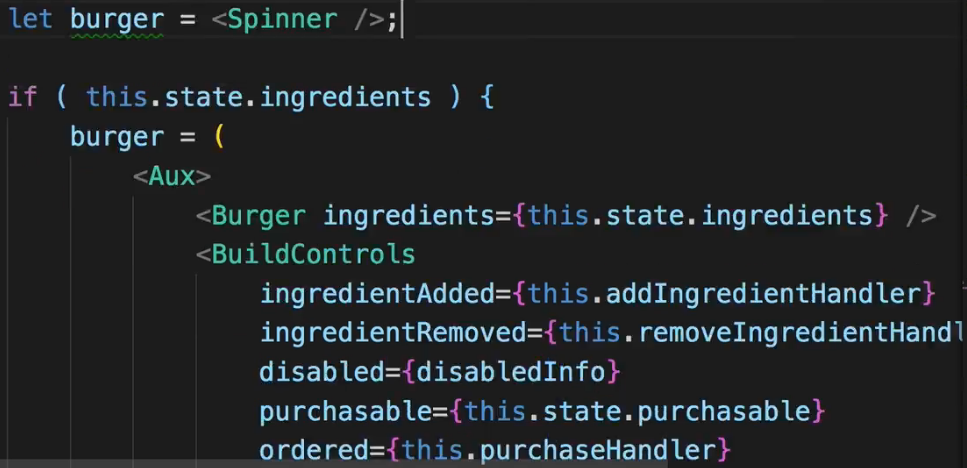
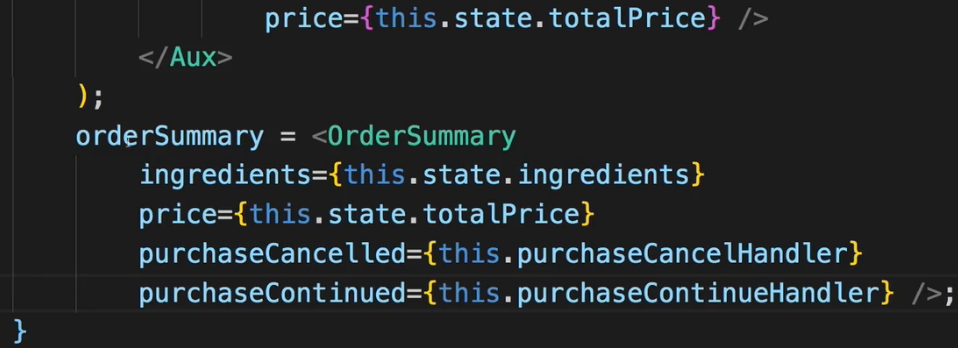
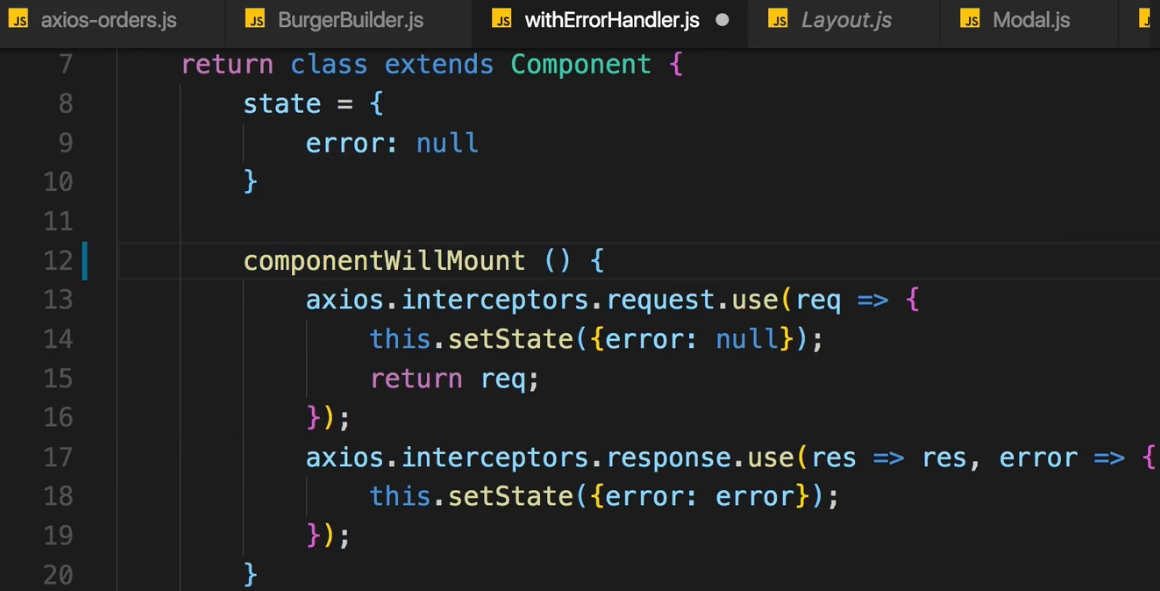
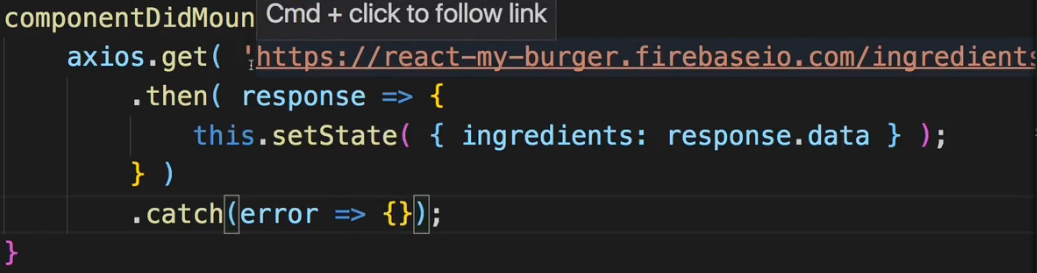
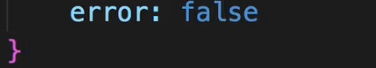
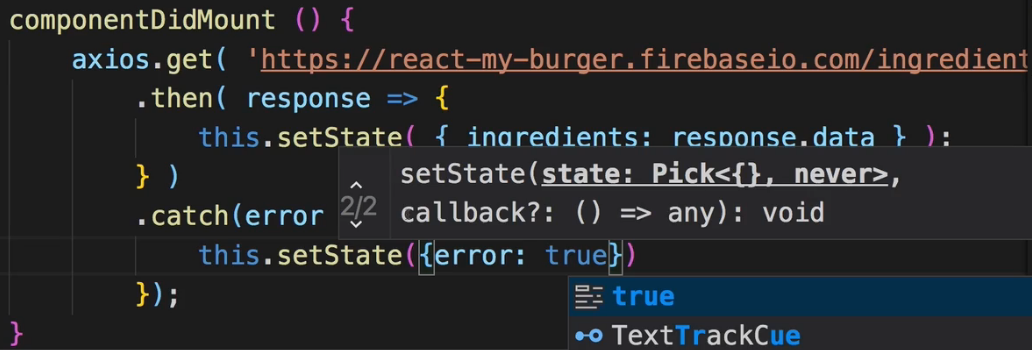
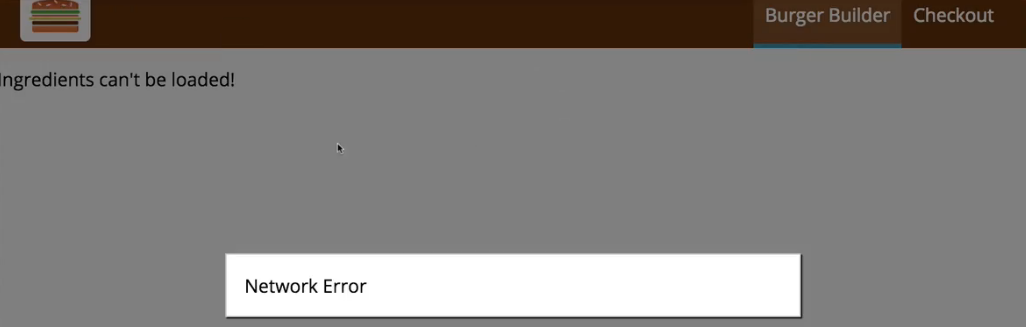
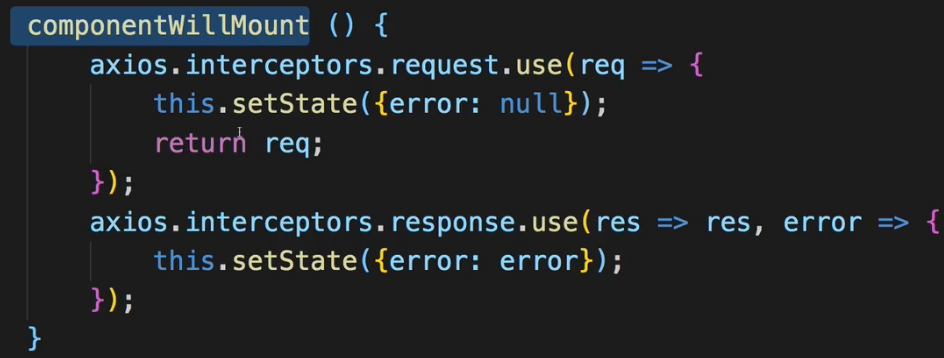
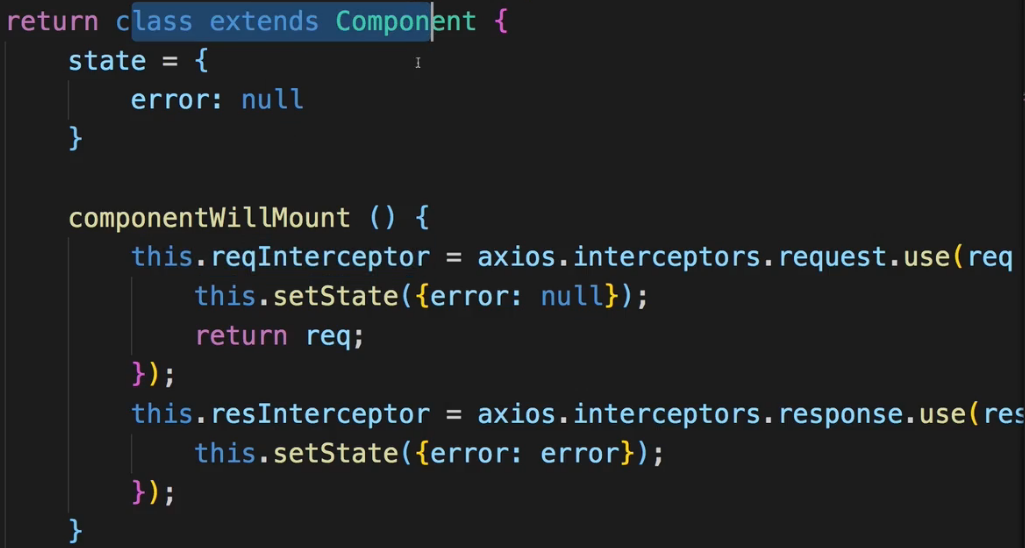
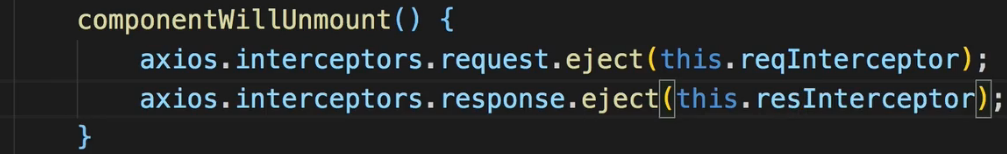
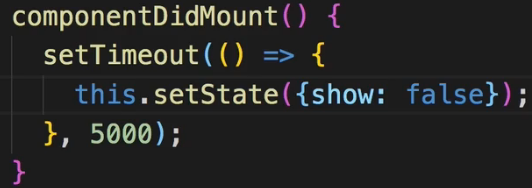
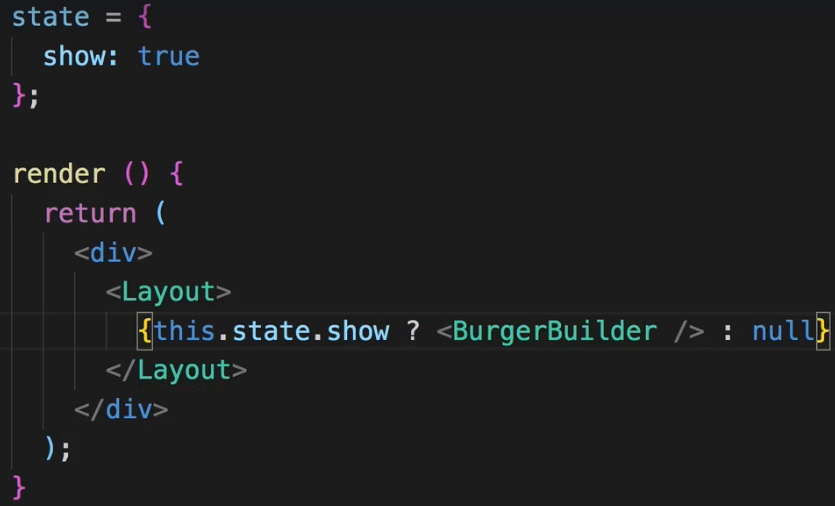
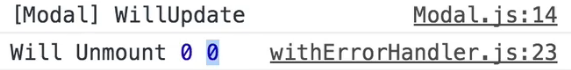
**Section 10 done: 9/9 Burger Builder Project: Accessing a Server  
Introduction**  
\* We can store that order the customer is about to place on a Server.  
\* Now we’ll fine tune this application later in the course once we know how to create different pages in our Single-Page app and therefore show a checkout form and so on.   
\* But for now storing the order and getting the orders, that would be an amazing advancement.  
=> We’ll implement a Back End so that we can actually store our orders so that we have a way of retrieving them even after our page was reloaded.  
=> For that storing on the Back End I’m going to use **Firebase**.  
=> Firebase is a service offered by Google, it’s essentially an out-of-the-box working Back End which includes a database. The cool thing is we can use it for free and for the amount of users in this project it’s going to say free. And we don’t have to write any Server-Side code.  
=> The way we communicate with Firebase is the same we communicate with any other Back End.  
**Creating the Firebase Project**  
\* <https://firebase.google.com/> => GO TO CONSOLE > Add project > react-my-burger > CREATE PROJECT.  
=> This will now configure a Back End project for you which automatically gives you some REST API Endpoints and a MongoDB-like database.  
\* Firebase has this syntax where the API Endpoints are directly mapped to Endpoints or Tables you could say in our Database.  
\* > Database > Realtime Database if you’re asked.  
  
\* Still we won’t be connecting to the database directly, we’re just sending HTTP Requests and Firebase is doing the mapping of HTTP Endpoints to the database.  
\* Now for this to be available, you have to go to RULES > set .read and .write to true.  


**Creating the Axios Instance**  
**npm install --save axios**  
\* **I will add it by setting an axios Instance but you can of course just import it globally and set these global defaults**.  
\* Creating axios-orders.js file.  
   
\* This is also the reason why I used this instance approach - because later in the course when we add authentication, we’ll also use a different URL with axios so I don’t want to set this baseURL as the global default even though for the moment I could do that.  
**Sending a POST Request**  
\* Later there will be a checkout form and so on but for now let’s just store the order.  
\* In BurgerBuilder.js:  
  
\* And a cool thing about Firebase is that it uses this MongoDB-like structure - we don’t actually have Tables here, we just have JSON-like nested structure.  
\* And if we send a Request to something like this url/orders, it’s going to create an orders node and store our orders beneath that node.  
\* **For Firebase ONLY - you have to add .json in the URL Endpoint**.  
\* **Just a little side-note: of course for your Production-ready application you should definitely calculate that final price on the Server because you probably have your products stored on the Server there - to make sure that the User isn’t manipulating the code before sending it and doesn’t manipulate the price**.  
\* Now let’s say for an order we also need some additional information which we can’t fetch right now because we don’t have a checkout form but which I still want to fill on the Back End with some dummy data.  
  
   
   
\* Now since we used the POST method, Firebase is automatically creating and managing a list here. And each list item simply gets a unique ID assigned and created by Firebase automatically.  
**Displaying a Spinner while Sending a Request**\* It’s a good practice to show something while sending the data - in case you have some latency etc.  
=> In BurgerBuilder.js:  
  
=> I want to show a spinner inside this Modal so instead of OrderSummary.  
\* Spinner component is clearly a UI component so we add `Spinner` folder inside `UI`.  
\* Now to creating a CSS Spinner we can take advantage of some resources we can find on the internet. > search for “css spinners”, you’re going to find “Single Element CSS Spinners - Projects”, which is a page with loads of CSS Spinners and the code for it which you can simply use.  
<https://projects.lukehaas.me/css-loaders/>   
=> You can also adjust them by picking a background color and then a front color.  
=> Create `Spinner.css` file and copy and paste the code there.  
 + I’ll rename .loader to .Loader just to be in line with other CSS files I use.  
=> Then grab the HTML code here on the spinner and use that in our Spinner.js component as JSX and the class needs to be adjusted to className.  
   
\* The Loading… inbetween is like a fallback in case the CSS isn’t displayed, then this will be shown.  
\* Now we need some condition - let’s add a new STATE.  
  
  
  
  
=> I want to stop loading even if the sending fails and even if an error occures.  
=> Of course I said I also want to close the Modal - well that can be done easily because the Modal is only shown if `this.state.purchasing` is true.  
=> So we simply have to set `this.state.purchasing` to false here.  
\* Now the spinner isn’t showing - why?  
=> We are correctly setting the loading and so on and orderSummary also is the Spinner therefore.  
=> But somehow our Modal doesn’t update and doesn’t show this when we set this.  
  
=> There is a good reason for that - consider our Modal.js code - we use shouldComponentUpdate() here and we basically only update the component if `this.props.show changes`.   
=> Here, however, the children of the component changed, so this.props.children changed. We’re passing a new child, we’re passing the spinner instead of the orderSummary, that doesn’t trigger an update here.  
=> **|| nextProps.children !== this.props.children**.   
\* Some styling seems to be broken, let’s fix this - in .Loader:before change the background color to white and .Loader:after also should be white background color.  
\* If you comment out the axios.post() part, you see the spinner for longer.  
**Handling Errors**  
\* Here again there are different places to where you could show such an error message.  
=> I simply want to set up some global Error Handler which shows a Modal with the Error Message but doesn’t use the Modal here in the BurgerBuilder.js because I don’t want to be stuck to this container - to the BurgerBuilder container.  
=> I want to have a flexible way of showing an error no matter in which component and which container it occurs.  
=> So therefore my Modal is going to get added to a higher level in the application and I want to have a higher level of the application so some higher component then this BurgerBuilder.js containr where I catch errors and where I then make sure that my Error Modal is displayed.  
=> I simple want to create a HOC with which I can wrap the BurgerBuilder or which I’ll actually use in that second way I showed you for using HOCs, so that we can simply wrap any component which should have this Error Modal with it and can conveniently add the Error Handling functionality to it.  
\* Creating `withErrorHandler` in `hoc` fodler - using this syntax/naming because I want to use this second way of creating a HOC which will be used here on the export and not used as JSX code.  
\* I’m using the props on the WrappedComponent because I don’t know them but I definitely don’t want to lose them.  
  
\* Using lower case withErrorHandler because we’re not going to use it as JSX.  
  
=> Now if `show` is true on the Modal, it will be shown. Now we only want to show this if we got an error. So we need to set `show` to something else and that something else needs to come from the WrappedComponent, we need that information if it did fail.  
=> To get that informatin, we should add a 2nd argument to our HOC - the axios instance so that we can set up a Global Error Handler on it.  
\* Now to use that instance here, I will change this Functional Component to a Class-based one.  
\* As you can see it’s an anonymous class - I don’t set up a name because I never use that class - I return it here, it’s a class factory essentially. withErrorHandler creates these classes.  
  
  
  
\* The 1st argument in the interceptor would be the response but we’re not interested in that -> null.  
\* We’re then setting our error to the error we’re getting here back from Firebase - and this happpens to be an object which also contains an error message on the message property, you can simply console.log() this object to see what it looks like.  
\* I’ll also set an interceptor for a Request and I’m not really interested in the request but there’s 1 thing I want to do in there - I want to call this.setState() and clear any errors so that whenever I SEND the Request, I don’t have my error set up anymore.   
\* There’s 1 more thing we have to do - the Modal we created also exposes the `clicked` property which occurs when we click the Backdrop and in this case I of course also want to clear the error because when we click the Modal, I don’t want to show it anymore, I have to get rid of the error.  
=> Now with this setup here, there’s 1 more thing I want to do - `**this.state.error.message**` which I’m outputting here, will throw an error initially because the Modal component is always present. Even if we don’t show it there.   
=> So what I’ll do is I’ll simply add a **ternary expression**.  
  
\* Now in BurgerBuilder.js I have to pass it the `axios` which we import there and which is our axios instance.  
  
\* Now with that, there’s 1 more thing we have to do - in our interceptors we of course have to RETURN something - when we’re sending the Request we have to return the request config so that the Request can continue and for the Response I also want to implement a response handler here, where I simply return the response.  
  
\* Now if we for example remove the .json from the Firebase Endpoint:  
  
\* Closing now doesn’t work because we set `clicked` instead of `modalClosed` property.  
  
=> So we change:  
  
**Retrieving Data from the Backend**  
\* In Firebase let’s manually create a new node ingredients and add something.  
  
\* Now on the BurgerBuilder I want to set up the state dynamically.  
\* And you learned that a good place for fetching data is **componentDidMount()**.  
\* Let’s set the ingredients in state to null initially, because we’re now fetching it from the database.  
   
=> Now we’re getting an error:  
  
=> Because now when we try to loop through our ingredients at the start of the app, that of course fails because initially ingredients is now null.  
\* We’re only fetching it at the start, and that’s a typical use case in applications - you fetch data when it loads.  
=> So parts of the UI which depend on the data will therefore fail.  
=> Now this of course can easily be prevented by checking if we have ingredients before rendering anything which depends on ingredients. Like the Burger here.  
\* Now there are a couple of ways where we can handle this.  
=> I want to handle it here in my BurgerBuilder container, I want to show a Spinner instead of the Burger and instead of the BuildControls while I’m waiting for the ingredients to load.  
  
  
\* You can use our **Aux HOC** or **FRAGMENTS <> </>**.  
  
=> However, we’ll face another issue - the orderSummary/OrderSummary will also use the ingredients and therefore fail.   
=> Let’s initially set orderSummary to the Spinner and then set it to the actual code in the same place where we set the Burger.  
   
=> Of course we also want to make sure that we overwrite this again if loading was set.  
=> So after this check where we check for ingredients, I’ll add this old if statement, where we check for the loading state to overwrite orderSummary when needed.  
  
\* Now if we load our application you can see that Spinner for a fraction of a second.  
\* The cool thing now is that if we go to Firebase and we update our ingredients so that for example we always start with meat, we can simply set meat to 1 in Firebase and now in our app we already start with 1 meat.  
\* Now another cool thing is that we’re showing the Spinner but if we now are removing the .json end in the URL where we’re getting the ingredients - just as an example - if we’re removing that and we go back, we don’t get the error Modal. Now that’s not cool - why are we not getting that? That can be hard to spot.  
=> The reason actually can be found in our withErrorHandler function - we’re setting up our interceptors in the **componentDidMount()** Hook and it worked great for the POST Request.   
=> But think about that Lifecycle diagram - there you see **componentDidMount() is actually called AFTER ALL CHILD COMPONENTS HAVE BEEN RENDERED => which means after componentDidMount() was completed in the child components**.  
=> Now think about our withErrorHandler - here we’re wrapping this WrappedComponent - which is our BurgerBuilder container for example. And that of course has 1 implication - componentDidMount() in the withErrorHandler will only be called once componentDidMount() was called here in BurgerBuilder. And since we reached out to the web in componentDidMount() of the WrappedComponent (BurgerBuilder), we never set up our interceptors.  
=> The fix is simple - we’ll **use** **componentWillMount()** **in withErrorHandler**  
  
=> **This will be called BEFORE the CHILD COMPONENTS are RENDERED**.  
=> **And we’re not causing Side-Effects here, we’re just registering the interceptors and we wanna do that before the child components are rendered**.  
\* Now if we reload the application (still with the .json missing), we get that error Modal but then we get another error that we fail to set our state.  
  
=> The reason for this is that the **.then()** block gets executed in our BurgerBuilder even though we have an error.  
=> Now the reason for that simply is that we’re not having a **.catch()** method here in **componentDidMount()** of the BurgerBuilder.js.  
  
=> **If we .catch() an error there and simply don’t do anything with it for example, then we’ll already not get this error anymore. Because now we’re not calling the .then() block anymore**.  
\* We can still dismiss the network error and our application is now broken, so we probably would want to handle that specific error case here for this specific page by for example also settings the UI here.  
  
  
=> Adding an error to the STATE.   
=> And then inside componenDidMount(), if we catch an error, we set error: true.  
  
\* Now with that we can go down to the render() method and we want to output an error message for this case where our application because unusable.  
\* Now here we have the burget set to the Spinner - we actually don’t want to show the Spinner but our “This application is not usable at all” error message if we have the `this.state.error` true. Otherwise we’re showing the Spinner or we’re overwriting it with the ingredients if we got them.  
  
  
\* Now our application is broken but at least we show it to the user.  
\* Of course once we fix that URL by adding .json, now everything is working.  
\* Now we got a really robust application where we’re both storing orders and fetching content.  
\* Now of course we could also implement code to fetch the orders from our database.  
\* So that is something we wanna work on next - ROUTING, being able to switch pages even though technically we only have one page in that Single-Page application.  
**Removing Old Interceptors**  
\* There is 1 improvement or FIX we should apply to our app though.  
\* It can be subtle to spot but it can really improve our application.  
\* Let’s look at this withErrorHandler HOC.  
=> There we obviously added our interceptors in componentWillMount(). And of course it makes sense to add them there because we wanted to make sure that we can intercept errors and hence show this error Modal we set up in this HOC.  
=> We have 1 issue though - this withErrorHandler method can be wrapped around multiple components - that’s the whole idea of having this HOC, otherwise we could’ve put that code directly into the BurgerBuilder.js if that were the only component where we want to use that. Right now it is the only component where we’re using it but again our HOC is of course not limitted to the BurgerBuilder.js.  
=> The problem we have is: if we add this withErrorHandler HOC to other components, we’ll call componentWillMount() again and again of course because the class component we return in this HOC, is created every time this withErrorHandler is wrapped around an existing component - like we do it on BurgerBuilder.js - so every time we call withErrorHandler on the element we’re exporting as we do in the BurgerBuilder.  
=> So we’re actually attaching multiple interceptors in our application and we’re attaching them to the same `axios` instance.  
=> The problem of course is and we don’t have it right now but we will face it later in the course (the mentioned ROUTING will lead to that problem) once we have more pages where we might use withErrorHandler - we of course create this instance multiple times - this “class extends Component” here and therefore all the old interceptors - so all the interceptors we set up when we wrapped this around another component which might not be needed anymore, still exist.   
=> So we have a lot of dead interceptors sitting in memory which actually are not dead but which still react to our Requests and the worst case - they lead to errors or do somehow change the STATE of our application - but even in the best case they leak memory because that’s code that still runs that is not required anymore.   
  
=> So we should actually remove the interceptors when this component gets unmounted. So when this specific instance of our withErrorHandler wrapped is not needed anymore.  
=> **componentWillUnmount()**.  
\* This is executed at the point of time a component isn’t required anymore.  
\* Now to be able to remove an interceptor here, we need to store a reference to the interceptors we create, in PROPS of this CLASS. We can simply add new properties.  
  
  
\* Now to quickly see if that works, I’ll go to the App.js file which is where we add the BurgerBuilder and I want to add some code to remove it after some time just to see if **componentWillUnmount()** gets called.  
=> We add a console.log() to the componentWillUnmount() to see that.  
\* And in App.js to make sure that BurgerBuilder disappears after a while, I’ll temporarily add a STATE to my App.js component - it is a Class-based Component so I can of course do this and I’ll add a `show` property which is true initially and I will then use it here to show the BurgerBuilder.  
  
\* And also just for this testing we set it to false in componentDidMount() using setTimeout().  
\* All this code we added: console.log(), state in App, componentDidMount and the ternary operator are only added for this testing, we’ll remove it after.  
\* We also add the this.reqInterceptor and this.resInterceptor to the console.log().  
  
=> We see 0 0 and that’s actually correct because these are simply the IDs - there are IDs being kept in memory by axios for both the request and response interceptors so 2 different lists of interceptors and they start with index 0 - we added 1 to each list: 1 request interceptor and 1 response interceptor so that is why we see 0 0 here.  
=> So it does work, it does successfully get the reference to the interceptor and we can rely on **.eject()** working and this will just ensure that whenever we don’t need the BurgerBuilder component anymore, and of course this is not the case right now in the application but will soon be, that we clean up the interceptors which we attached due to using withErrorHandler on the BurgerBuilder so that if we reuse withErrorHandler in our application, we don’t create more and more interceptors with the old ones living on.   
\* Now we can remove all that testing code we added.s