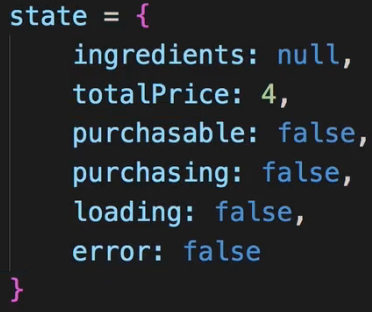
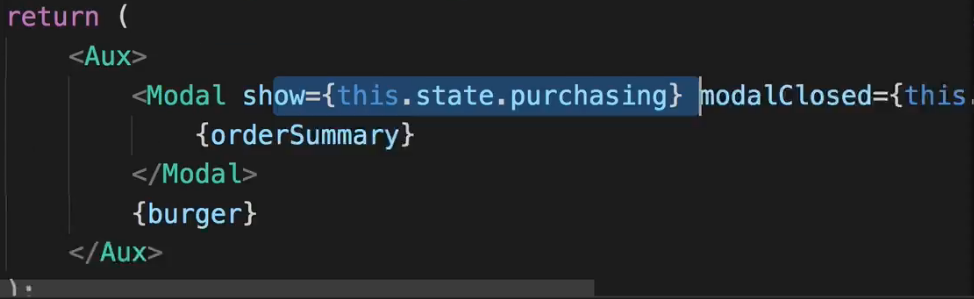
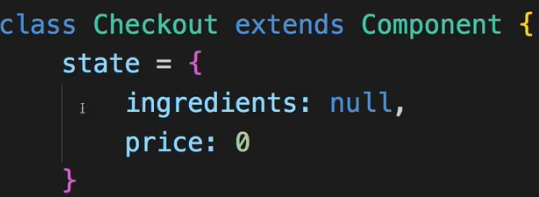
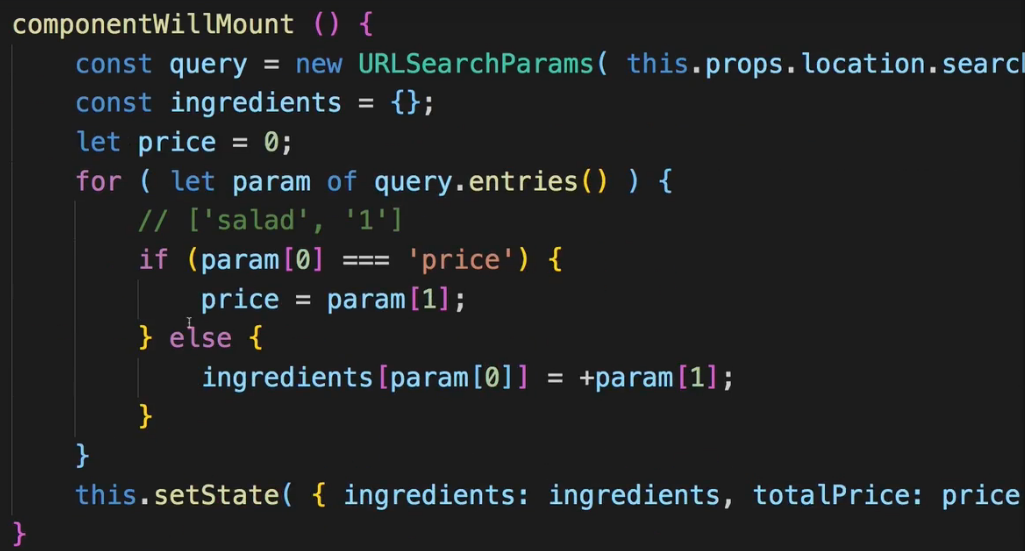
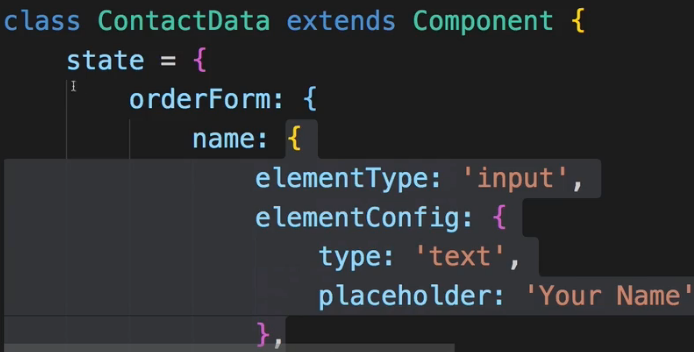
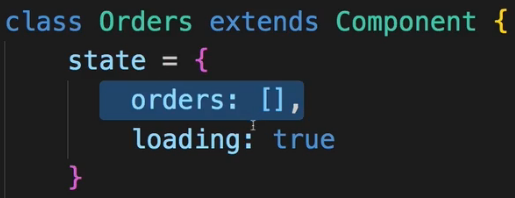
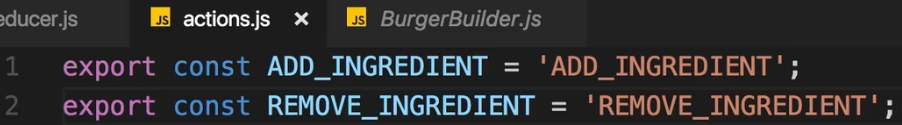
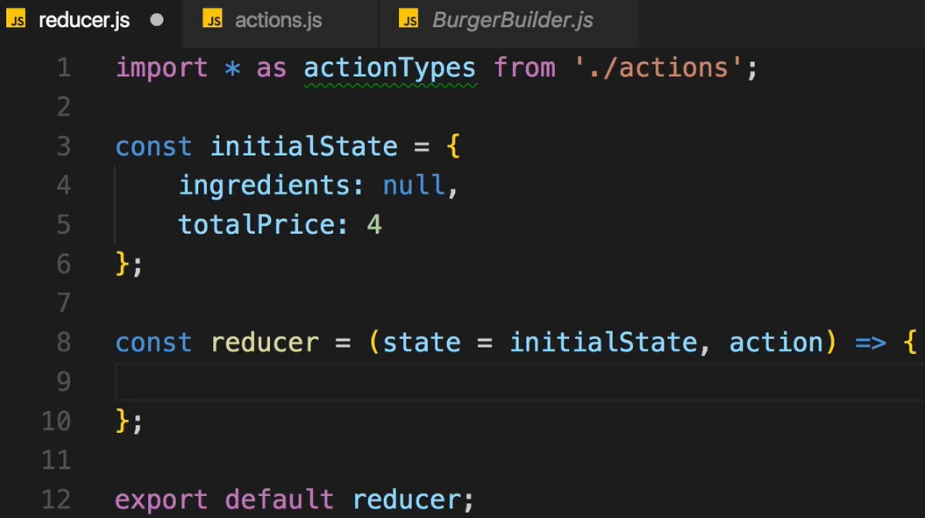
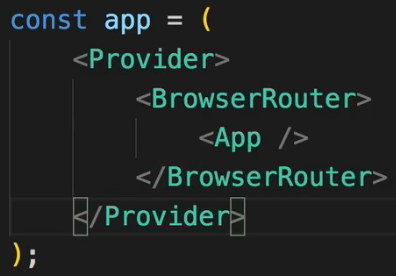
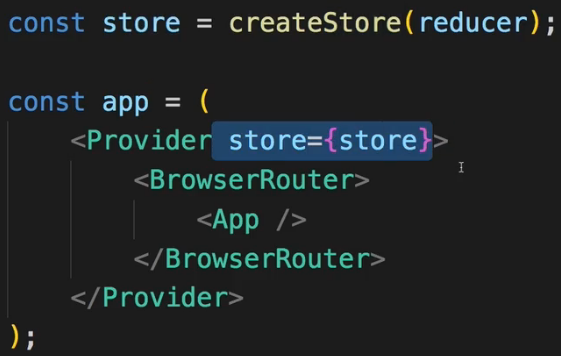
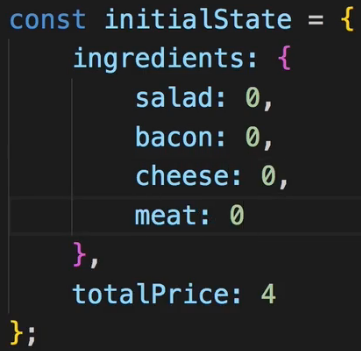
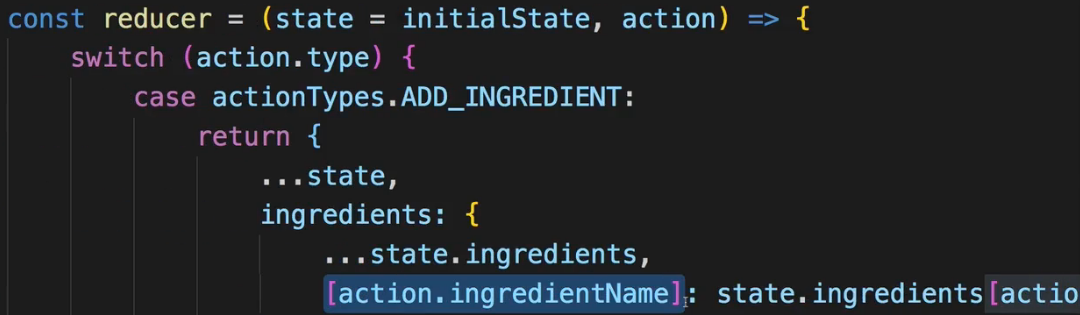
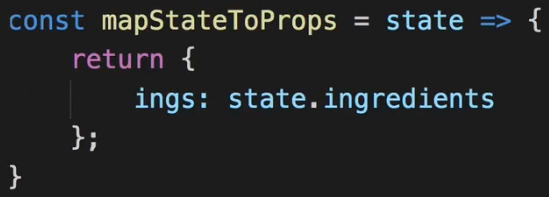
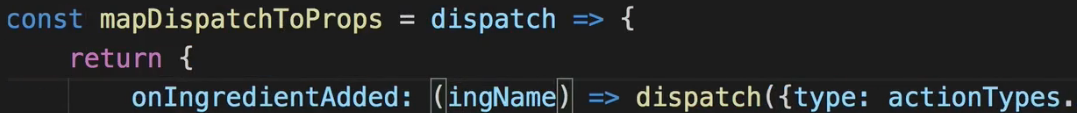
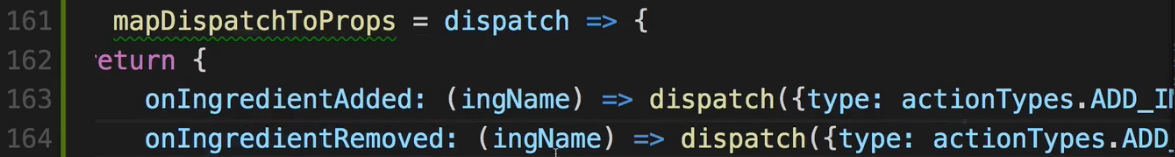
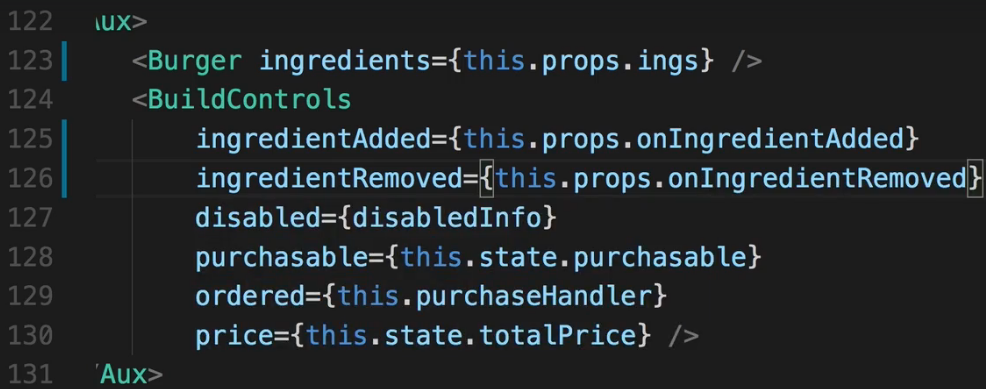
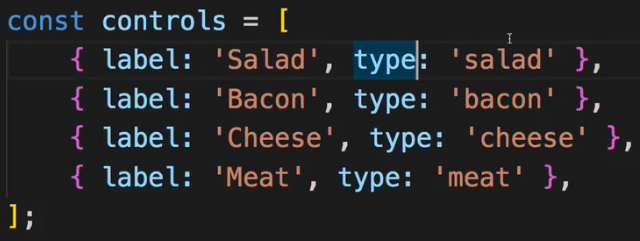
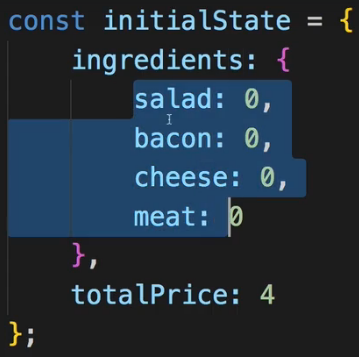
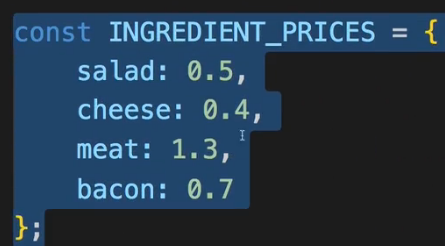
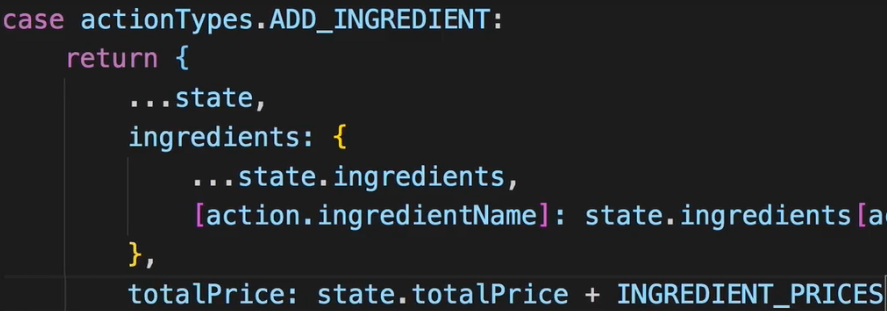
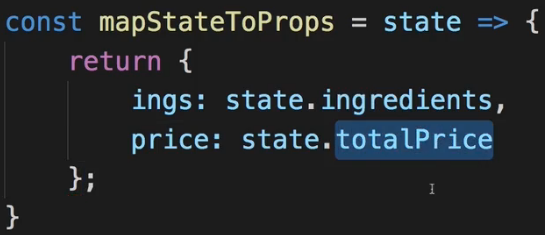
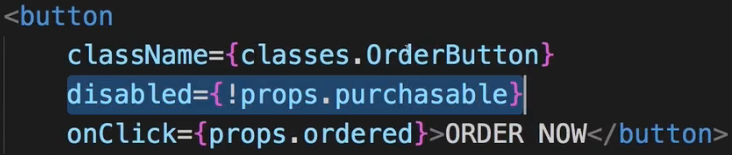
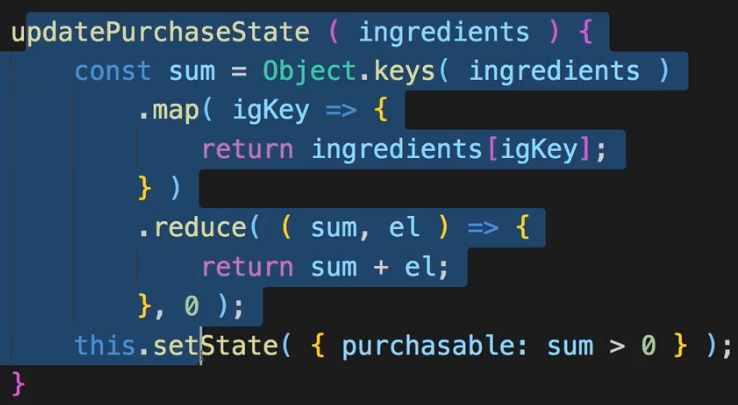
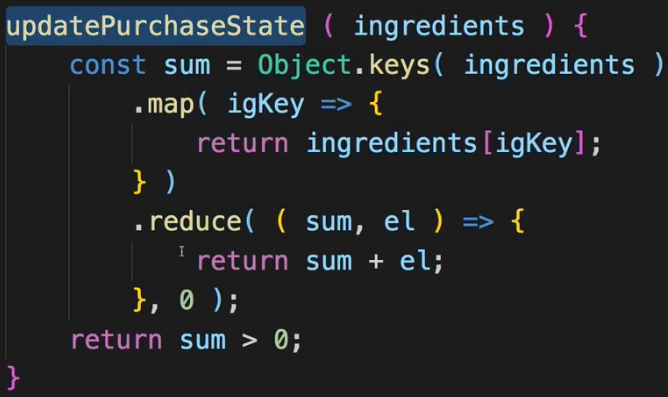
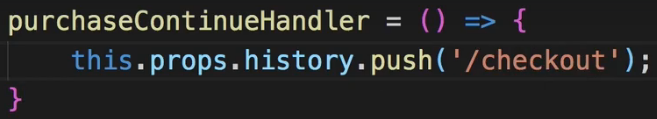
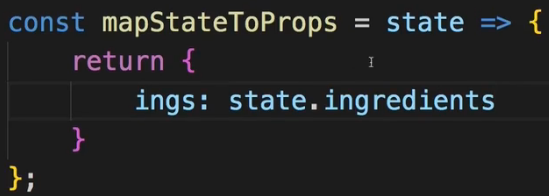
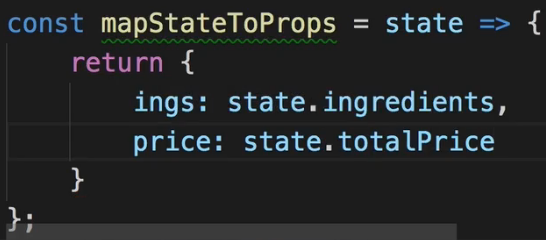
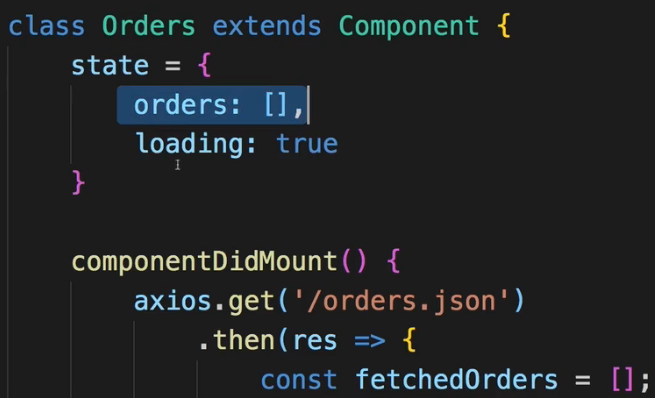
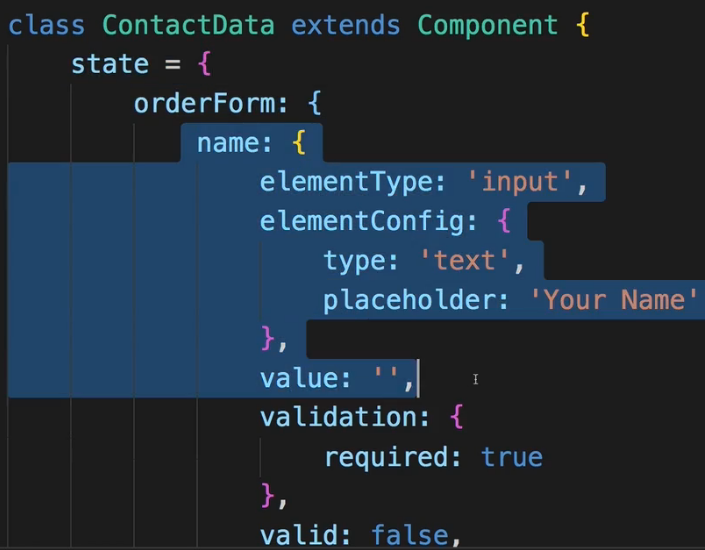
**Section 15 done: 10/10 Adding Redux to our Project**  
**Introduction**  
\* Let’s see where in our course project we can add Redux to make state management easier.  
\* Let’s dive into our containers because thankfully we already have a structure where all state management takes place in containers and therefore we don’t need to dig through all the components. It’s really just the state we manage in containers we probably want to manage through Redux in the future.  
\* Let’s start with the **BurgerBuilder.js**.  
  
=> We got the ingredients, the price and then a couple of state fields regarding the current status we have when it comes to purchasing that burger.   
=> `purchasing` we use to conditionally show or hide that Modal.  
  
=> `loading` is used to display a Spinner.  
=> `error` is also used here to render different content.  
 or spinner  
\* So if we have a look at our state, we could argue that `purchasing`, `loading` and `error` are kind of Local UI State. We use them here to determine whether we show a Modal, whether we show an error message. We could of course also manage that through Redux and I won’t say that this would be bad, not at all, you can definitely manage everything through Redux - but there also might not be a necessity to do so.  
=> So what’s definitely interesting to manage through Redux though is the `ingredients` and `totalPrice` of the burger.  
=> `purchasable` we pass to our BuildControls and there to disable/unlock the order button. So this also is more the UI, we change something on the UI, it might not be super important for us to manage that through Redux, though of course you could say `purchasable` - the value it takes - depends in the end on the burger we configured, so we’ll have to see where we add this.  
\* The `ingredients` and `totalPrice` are definitely a Redux case though.  
\* Let’s have a look at the **Checkout.js**.  
  
\* We also have `ingredients` and `price` here - this already is a strong case for using Redux because here we have that issue of passing the `ingredients` through query params. It would be awesome if we could get rid of that.   
  
\* It would also be awesome if we in the render() method could change the way we render this Route so that we don’t have to use this way just to get the `ingredients` to ContactData.  
  
\* Instead, once we do actually manage the `ingredients` state in our Redux STORE, we can just render this component here and in **ContactData.js** we can CONNECT this through the CONNECT HOC to the Redux STORE so that we get the `ingredients` here too. Because in the ContactData component, if we have a look at the state:  
  
=> We of course got our orderForm - all these controls, and I want to leave them here because that again is Local UI State, we don’t really need information about our form here anywhere else in the application. But we also of course do take advantage of the ingredients which we get through props so we don’t seem them in the state here.  
\* So in the ContactData the state is really just all the UI State here, handling the form, but we get some props here which we in the future might directly get from the Redux STORE instead of passing them here with this Route workaround we’re currently using in the Checkout.js container.  
\* Now in the **Order.js** component, there we also have the `orders` in the state.  
  
\* Now the `orders` are actually fetched from the server and we haven’t learned how to fetch data from a server, how to handle asynchronous ACTIONS, as HTTP Requests of course are, together with Redux, so `orders` is something we’ll take a look at later in Advanced Redux.  
**Installing Redux and React Redux**  
**npm install --save redux react-redux**  
\* Add `store` folder in the `src` folder.  
\* Add `reducer.js` and `actions.js`.  
\* Let’s start with the basic ACTION TYPES we’ll probably need.  
=> For that we can have a look at the methods we created in BurgerBuilder.js.  
=> **addIngredientHandler**, **removeIngredientHandler** - these are the 2 main things we do.  
  
  
\* Let’s set up the state management in `index.js` file where we wrap our entire application.  
\* There we see something interesting - we already do wrap our <App /> with the <BrowserRouter> that of course can lead to confusion,it is something we haven’t looked at before - where should we now add our Provider component from react-redux? Inside or outside the <BrowserRouter>?  
**Basic Redux Setup**  
  
=> The **Provider** **should wrap everything.**  
  
\* And there is something special about using react-redux with React Router and I’ll come back to that and how we fix this - it basically has to do with making sure that the CONNECT functionality and the ROUTING functionalities work together fine because both implicitly set up some props on the wrapping component and we have to make sure that everything works there.  
\* For now let’s create the STORE.  
  
  
  
**Finishing the Reducer for Ingredients**  
\* Let’s simply add a default: case to the switch statement instead of returning it outside this time.  
  
\* Now in the state we start with `ingredients`: null but keep in mind that in the BurgerBuilder we start with null because in componentDidMount we actually load our ingredients, our starting ingredients (usually all 0) from the internet. And this is something I’ll re-add but for now since we haven’t learned how to handle asynchronous code here, I will comment out the code in componentDidMount() and instead start with my `ingredients` set up in the state in the `reducer.js`.  
  
=> Now let’s add an ingredient immutably as always.  
=> First distributing the properties of the state and then also distributing the properties of `ingredients` so that we really do it immutably, simple using …state doesn’t create deep clones of objects, it doesn’t go into objects and create new ones for these too, it simply just copies the surrounding object but `ingredients` which is an object on its own would still point to that old object so we fix it by spreading out the properties of that old object into a new object. And now we want to overwrite the given ingredient which we get as a PAYLOAD of this ACTION.  
\* Now in ES6 there’s still special syntax you can use to dynamically overwrite a property in a given JavaScript object, you can use square brackets [] now this doesn’t create an array here, instead here you can now pass a variable or something which contains the name you actually want to use as a property name. And I expect to get that property name on my ACTION. So one of the ingredients, whichever it is in the PAYLOAD `ingredientName`, will receive a new value.  
  
\* Now for removing I also want to create a new version of my state where I simply decrement 1 ingredient by 1.  
=> It’s the same as adding, but -1 instead of +1.  
**Connecting the BurgerBuilder Container to our Store**  
\* It works with the **connect** function we import from react-redux.  
  
\* Let’s wrap our export with it, now to be precise, let’s not wrap it with it, let’s simply wrap the mapDispatchToProps and mapStateToProps constants with it and pass the BurgerBuilder to it.  
\* But here we face an issue:  
  
=> We already have a HOC wrapping our export.  
=> Well, this actually isn’t a problem, you can have as many HOCs in there as you want.  
=> In the end what connect will do is it will just set some props on the component it’s wrapping so as long you pass …this.props on in your own HOCs, this should work fine because any props set by other HOCs which might wrap this one, will still be passed on just fine.  
  
  
\* We need to pass the `ingredientName` along with the TYPE, that’s how we set it up in REDUCER.  
\* Now the `ingredientName` is something I expect to get here in this function so I’ll simply name it `ingName` here.  
  
\* We do the same for removing.  
  
\* Let’s use it to connect now.  
  
=> Let’s pass this entire `withErrorHandler()` call here as an argument to the function that this connect() function call returns us.  
\* Now we have our BurgerBuilder connected to the STORE.  
=> Let’s make these props like the `ings` available.  
=> **Everywhere where we used `ingredients`, we now want to use `ings`. And we can remove `ingredients` from the Local State**.  
=> For example instead of `this.state.ingredients` we now use `this.props.ings`.   
\* Now let’s work on the addIngredient and removeIngredient handlers. We instead execute `this.props.onIngredientAdded` and `this.props.onIngredientRemoved`.  
  
=> **BUT both functions need an ARGUMENT in the end (ingName)**.   
=> To pass this on here, let’s first of all have a look at our BuildControls component.  
  
=> We see that we already pass `ctrl.type` here as an ARGUMENT to the `props.ingredientAdded` and `props.ingredientRemoved` methods so to the methods which get passed through these props, and that happens to be the methods we just set up here in BurgerBuilder.  
=> Now that just means that we have to check if a `ctrl.type` is exactly what we’re looking for and it is, because `ctrl.type` is this:  
  
and that has the same format as the names we use as properties in our REDUCER code. And of course we should make sure that this is equal.  
\* You could of course go so far and refactor your application to take the `ingredients` state - the property names here:  
  
to create your `controls` in the BuildControls component but we’ll leave it as it is.  
\* The ingredients are now working great in our application.  
**Working on the Total Price Calculation**  
\* The price changes whenever we add/remove an ingredient.  
\* Now there are 2 routes you can take:  
=> 1. => you can define a new ACTION TYPE for the price change. You could call it UPDATE\_PRICE.   
=> And then we could also of course bind this in the BurgerBuilder at the bottom to a property which we execute to dispatch this ACTION and this ACTION would simply be called by the REDUCER and there we calculate the price. This would be fine and it would ensure that every ACTION leads to just 1 piece of your state being manipulated in your REDUCER.  
=> 2. => On the other hand, we already got the only 2 ACTION TYPES which change our price. ADD\_INGREDIENT and REMOVE\_INGREDIENT. So it would also be fair to not only update the `ingredients` here but also update the price.  
\* Do you only want to change 1 of your state per case? Then you need an extra ACTION.  
\* Are you fine with manipulating multiple pieces of your state? Then you can take 1 ACTION.  
\* In BurgerBuilder at the top we got the prices:  
  
I’ll cut it from there and move it into my REDUCER. So there I will not only define my initialState but also my INGREDIENT\_PRICES.  
\* We could define this anywhere in our application, we could fetch this from a database or from a server, though right now we couldn’t because we don’t know how to handle asynchronous could but in the future we will.   
\* I’ll simply hard-code it in there.  
=> And there again we got the same ingredient property names as we use everywhere in our application so now we can simply set the totalPrice equal to the old totalPrice + and now access the INGREDIENT\_PRICES for the ingredient we get here in our ACTION, the `ingredientName`.  
  
\* For removing we do the same but with - instead of +.  
\* Now when we try adding ingredients, the Current Price doesn’t change, why?  
=> Well because we now do update it in our Global State through the REDUCER but of course in the BurgerBuilder.js container at the bottom we still only fetch the `ingredients` from our Global State. So we have to also fetch the `price`.  
  
\* We can now remove the totalPrice from the state there as well.  
\* **Let’s replace `this.state.totalPrice` with `this.props.price`**.  
\* Now the price gets updated correctly.  
\* Now 1 thing we still don’t see is that the ORDER NOW button would get enabled - it stays disabled. This is what we’ll work on next.  
**Redux & UI State**  
\* The ORDER NOW button never gets unlocked.  
   
=> The reason for this is that the property determining if it is unlocked or not, is the `purchasable` property we’re setting on the BuildControls.  
=> So we need to update the `purchasable` prop whenever our `ings` change.  
\* I can now remove the `purchasable` from the Local State.  
\* I leave this method here:  
  
to see the logic for enabling it.  
=> Now we have 2 options:  
=> 1. => We could leave the method and simply adjust it a tiny bit to return the result of our boolean check here and we still calculate the sum here with ingredients we pass to updatePurchaseState().  
  
=> We could then go down to our render() method and there where we pass `purchasable` and we could set it to our `this.updatePurchaseState` instead of `this.state.purchasable` which now simply returns true or false and return the result of this function call.  
  
=> **Here we need to EXECUTE this because we want to execute this whenever this gets re-rendered so that we fetch the updated result. And there we now need to pass the `ings` as the ARGUMENT**.  
=> 2. => Alternatively of course we could also get our ingredients through this `ings` property here, directly in updatePurchaseState() by not expecting an ARGUMENT but accessing `this.props.ings` here. It’s in the same class after all.  
\* Now the button gets enabled/disabled again.  
\* This is one way of doing this and it’s absolutely fine because you could consider this Local UI State.  
\* You could of course also say: No, this is something I want to manage in my Redux state, and you always have etch cases like this, and oftentimes there is no clear right or wrong, use whichever feels easier to you, which makes your application better to you.  
\* So you could add `purchasable` to the REDUCER state and also change that when ingredients change.  
**Adjusting Checkout and Contact Data**  
\* Now that we’re handling everything related to ingredients through Redux, we should also adjust the other components where we use that ingredient data. Namely Checkout.js and ContactData.js.  
=> Especially these 2 are super great now that we use Redux - because thus far we used query params to pass our ingredients to our Checkout.js component.  
=> This something we can finally get rid of.  
\* In BurgerBuilder we can remove all the query params code in the purchaseContinueHandler(). We still want to navigate on but I don’t want to pass query params anymore.  
  
=> We can simply get the ingredients from the Redux STORE.  
\* **In Checkout.js**.  
=> We can get rid of componentWillMount() in Checkout.js.  
=> We can get rid of our state there which only handles ingredients and price.  
\* And in Checkout.js import CONNECT because obviously we want to CONNECT this container with Redux too.  
  
\* We don’t need mapDispatchToProps because we’re not dispatching anything in this container. We just navigate a little bit but we don’t do this through Redux STORE, we do it through React Router.  
\* By the way if you only had mapDispatchToProps, pass null as the 1st argument.  
=> Now let’s change `this.state.ingredients` to `this.props.ings` and the same for the price.  
\* We only needed the price down here where we did our little trick for loading the ContactData. Now thanks to our Redux STORE we no longer need to use the tricks so we don’t actually even need the price.  
  
  
=> Now the ContactData component in Route is not included as a JSX component but just pointing to it on the component property of the Route.  
\* **In ContactData.js**.  
\* Here we also use the ingredients and the price. Prior to Redux we got it with this little trick we just removed. Now we can simply CONNECT this container here to Redux as well.  
  
  
=> Now we need to change the `this.props.ingredients` to `this.props.ings`. Or we could name it `ingredients` here in mapStateToProps.  
\* Now everything in our application works, we’re passing around our Redux state successfully, now without any fancy tricks, really taking advantage of Redux.  
\* And this also hopefully shows where Redux shines.  
\* This is not a super huge application but it is a bit more advanced than just having a to-do list.  
\* And here we really make it much easier to ourselves if we use Redux.  
\* We have a predictable Data Flow and we make passing data around in our application super super simple.  
**Wrap Up**  
\* In Order.js we don’t manage `orders` in the state with Redux right now.   
=> Because `orders` really is related to asynchronous code and I want to handle fetching data and handling asynchronous actions with Redux first in the next module before we also use Redux on this therefore.  
  
=> The same is kind of true for the BurgerBuilder.js where previously we fetched the `ingredients` from the server. We’re not doing this right now, we’ll re-introduce this over the next modules.  
\* But what we are doing is managing the core `ingredients` through Redux.  
\* What I also already mentioned is that for example in the ContactData.js we still have a lot of state in our state property.  
  
=> The whole form is inside there, the validity of our controls.  
=> And I want to really highlight: you CAN manage all of that through Redux. You can do that, you can dispatch an ACTION on every keystroke the user does and put all that logic you have in that container here, into Redux.  
=> That you check the validity of controls in Redux, that you update the form control values through Redux, that you have Redux for everything. You can absolutely do that.  
=> And if you want to do this really depends on the style you want to have in your application.  
=> Do you want to have very lean components and containers? Then you have to put everything into Redux.s  
=> Do you want to have the state management placed where it belongs? So say this really is Local UI State, it’s only relevant in this component and shouldn’t be affecting any other components? Then you might keep it in this container here as it is right now.  
\* So this is really something you have to decide.  
\* Both is correct.  
\* It’s certainly worth the exercise to put all that state into Redux with the knowledge you have thus far but I’m happy with the current state of the application.  
\* Now let’s dive into the some more advanced things about Redux - like handling asynchronous code and then coming back to this application to improve it even more.