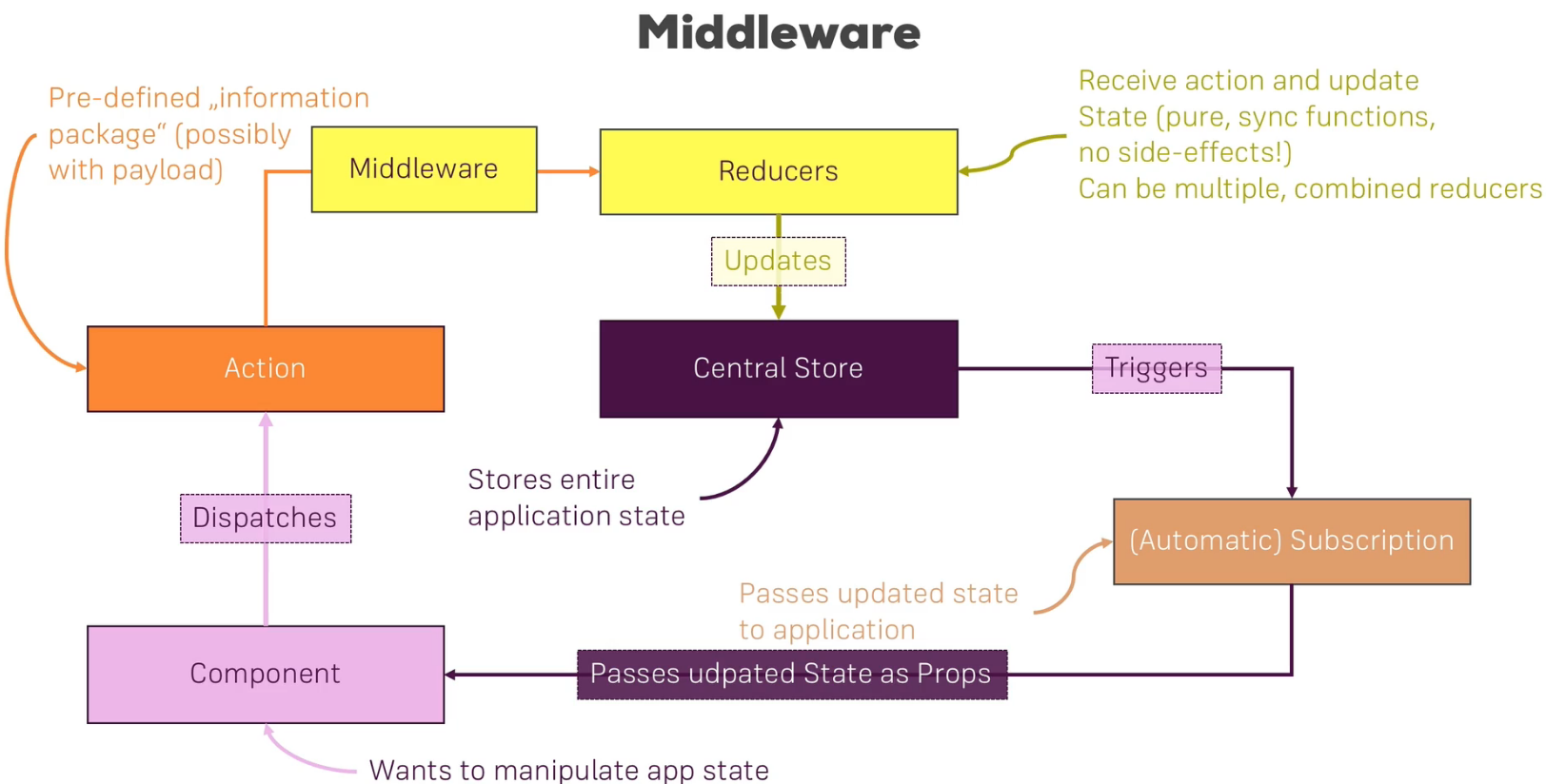
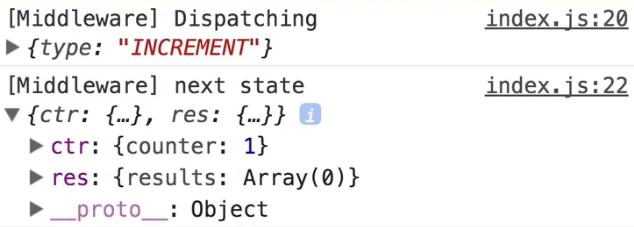
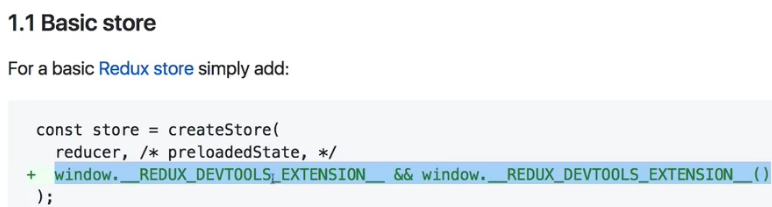
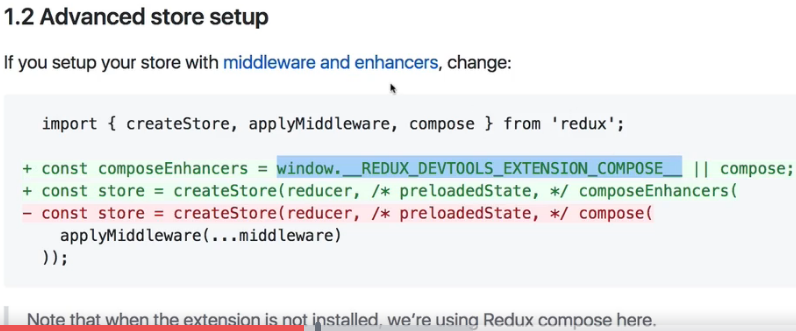
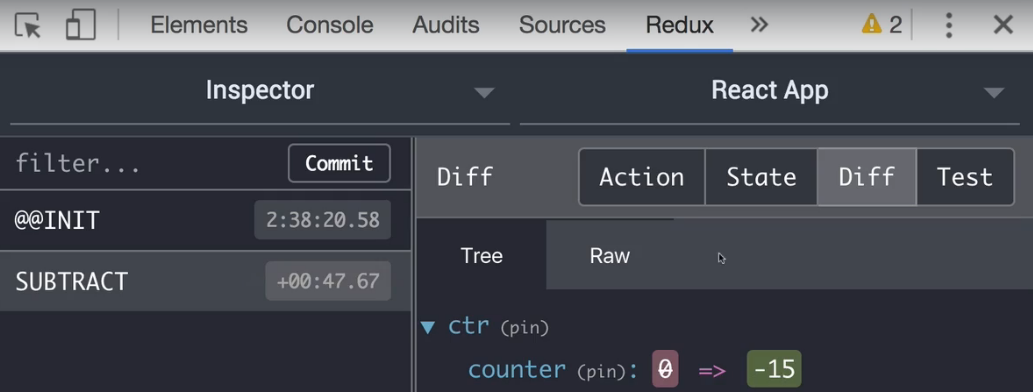
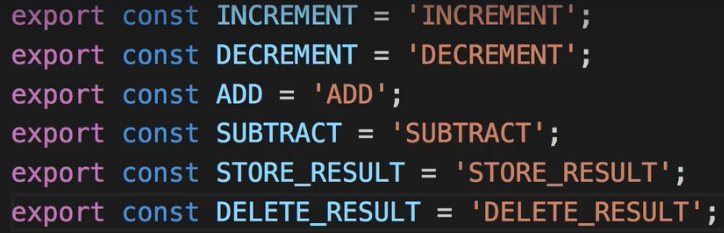
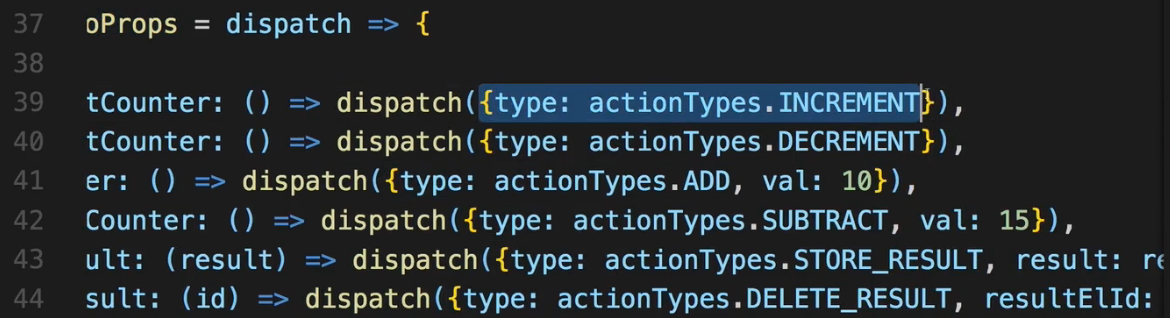
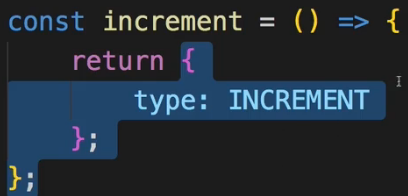
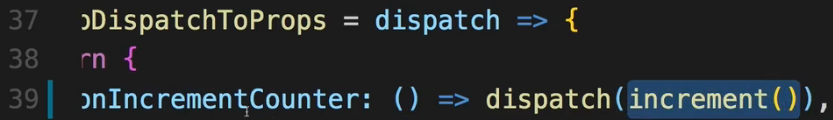
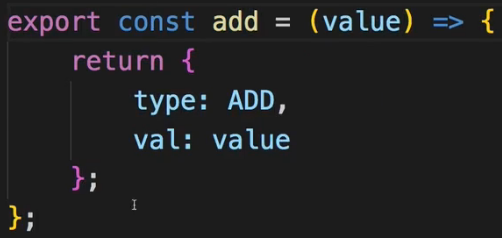
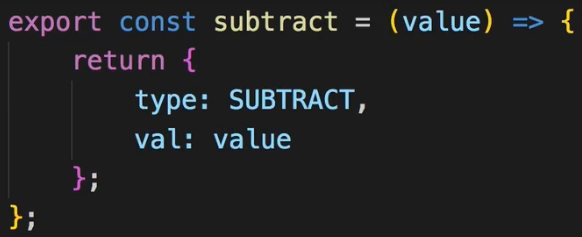
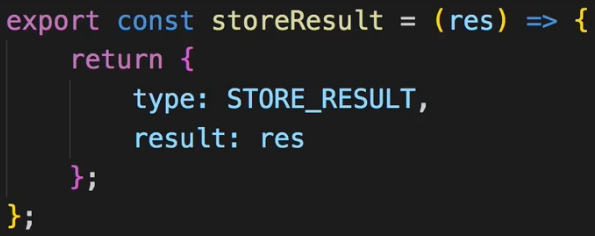
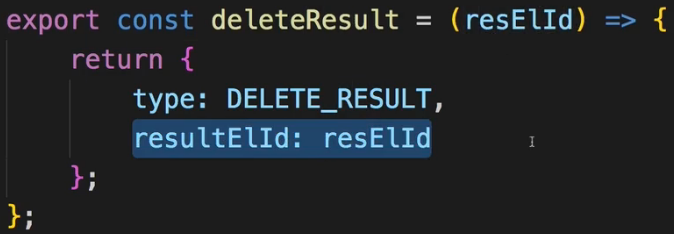
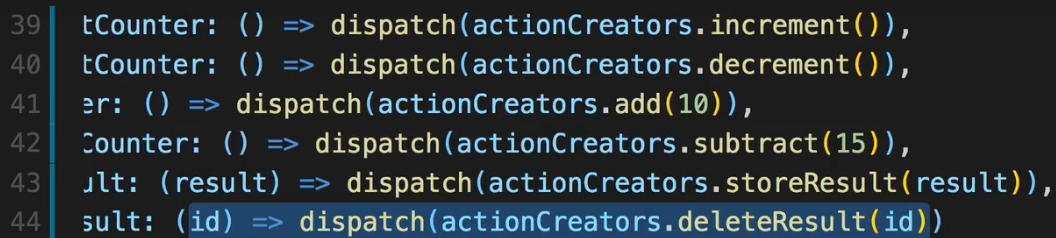
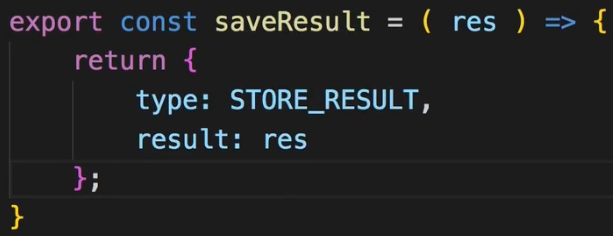
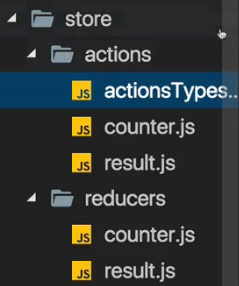
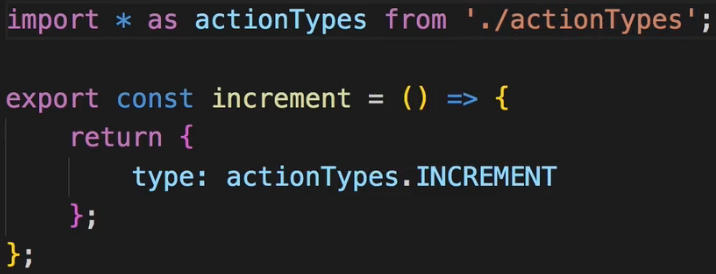
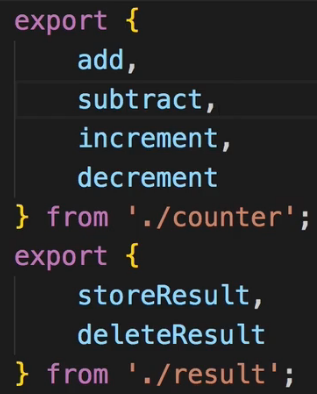
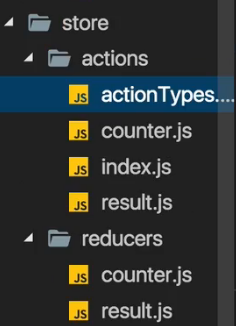
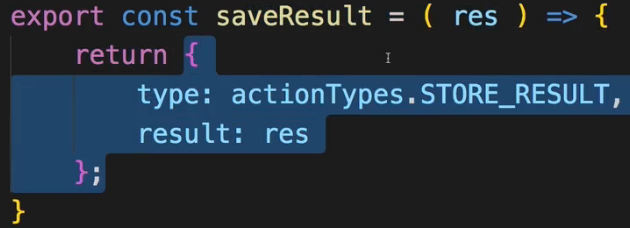
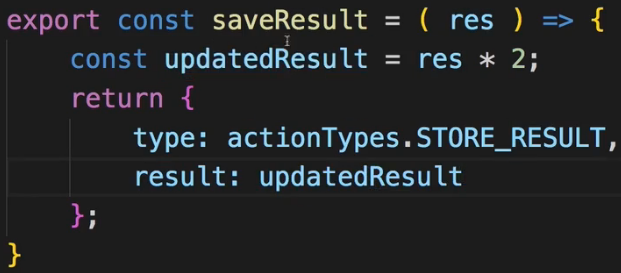
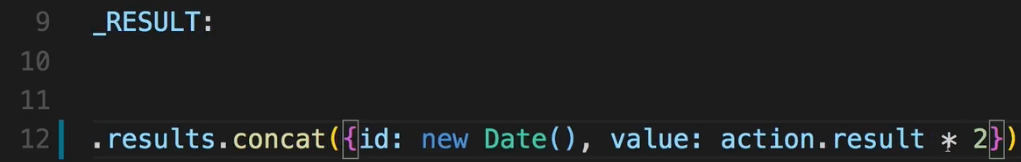
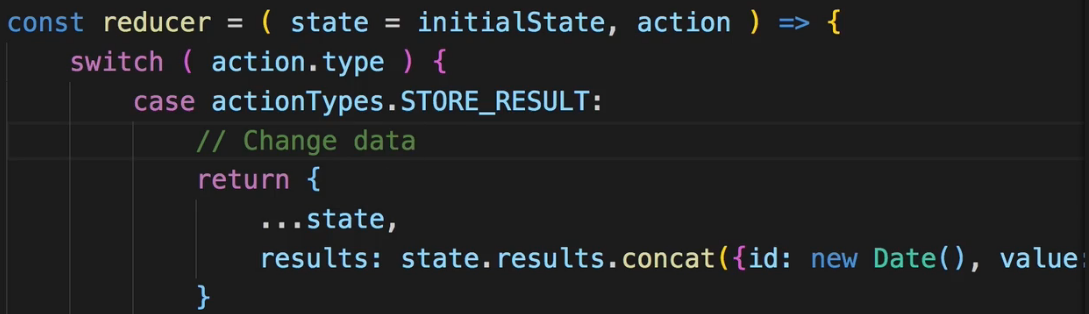
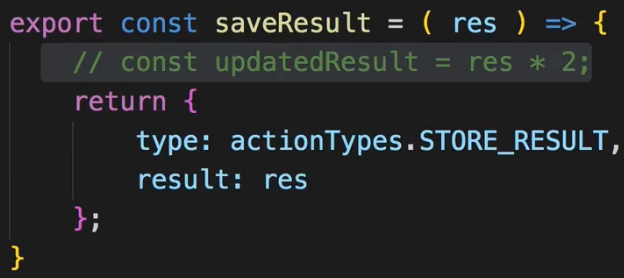
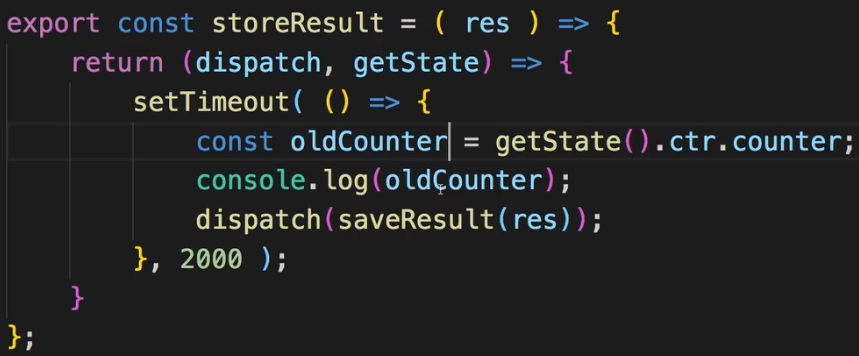
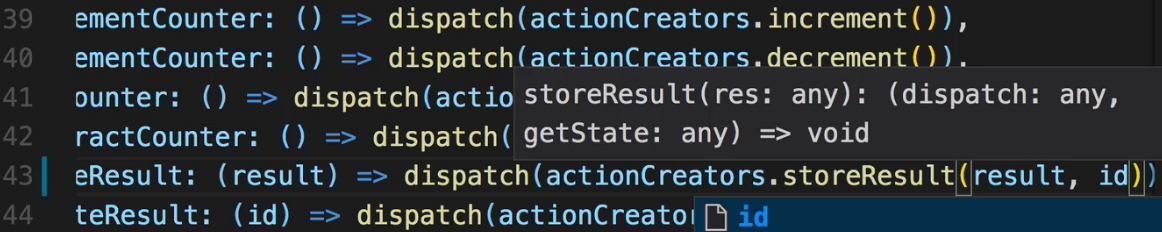
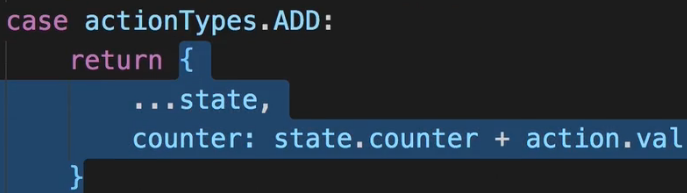
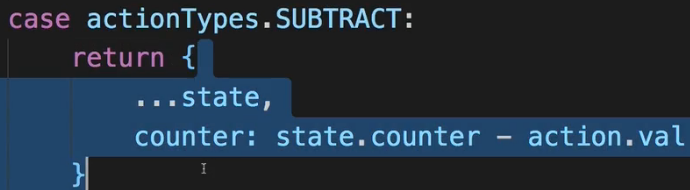
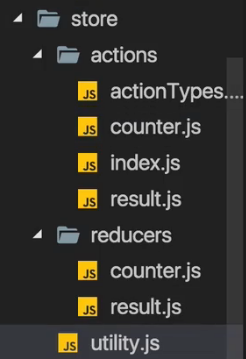
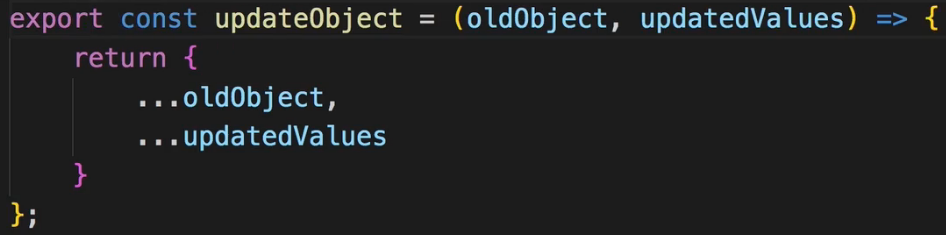
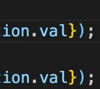
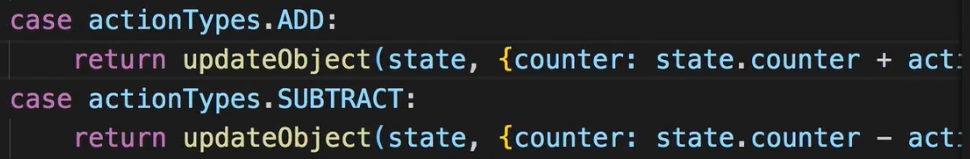
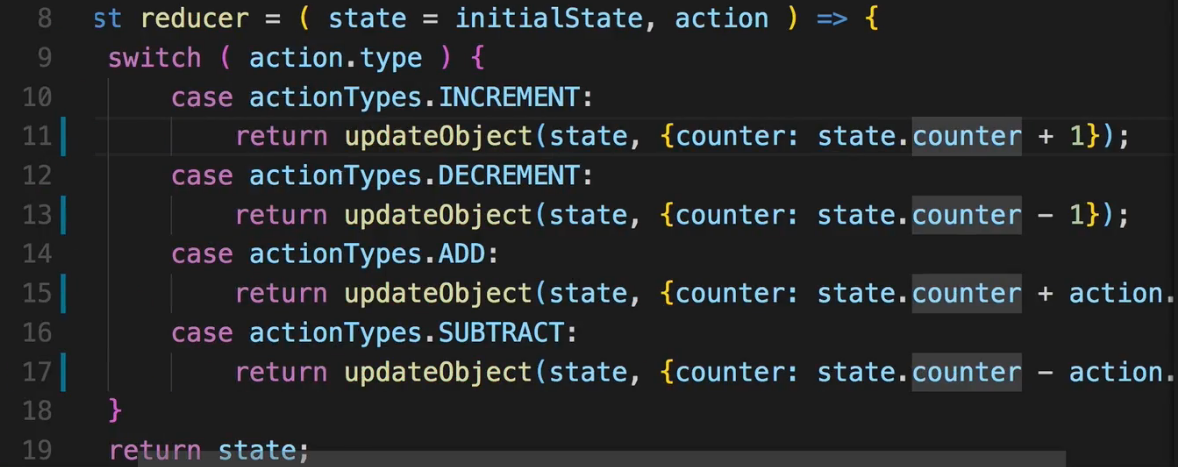
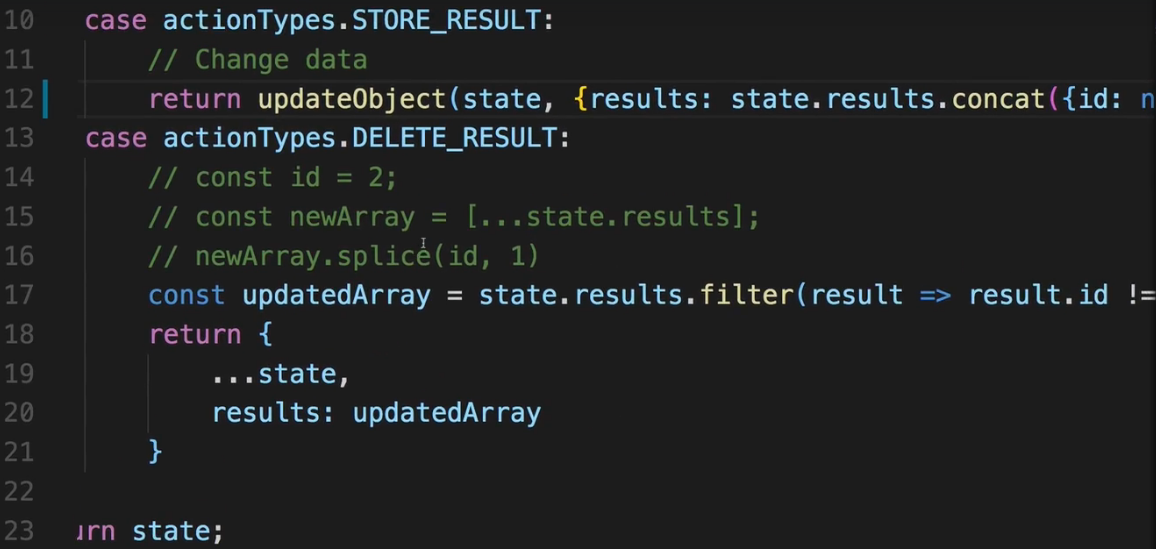
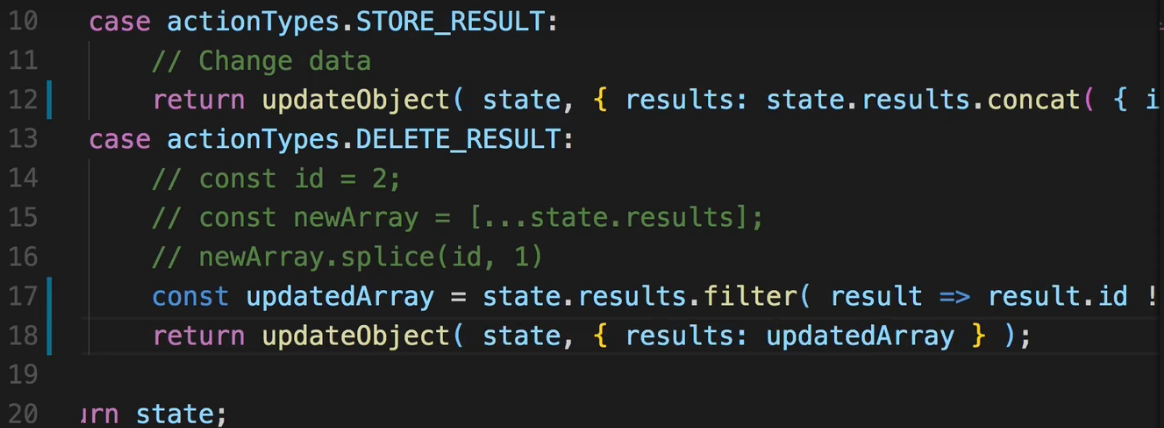
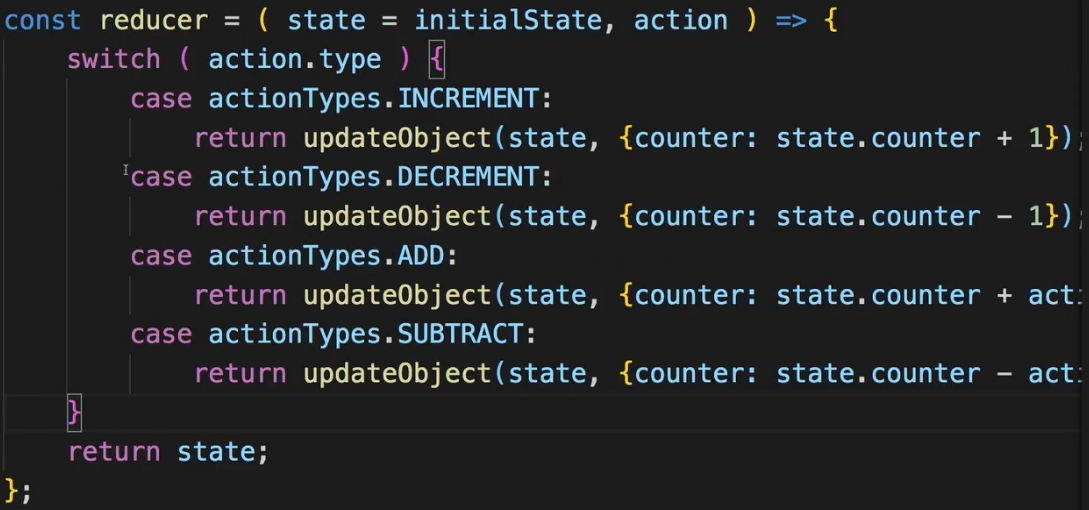
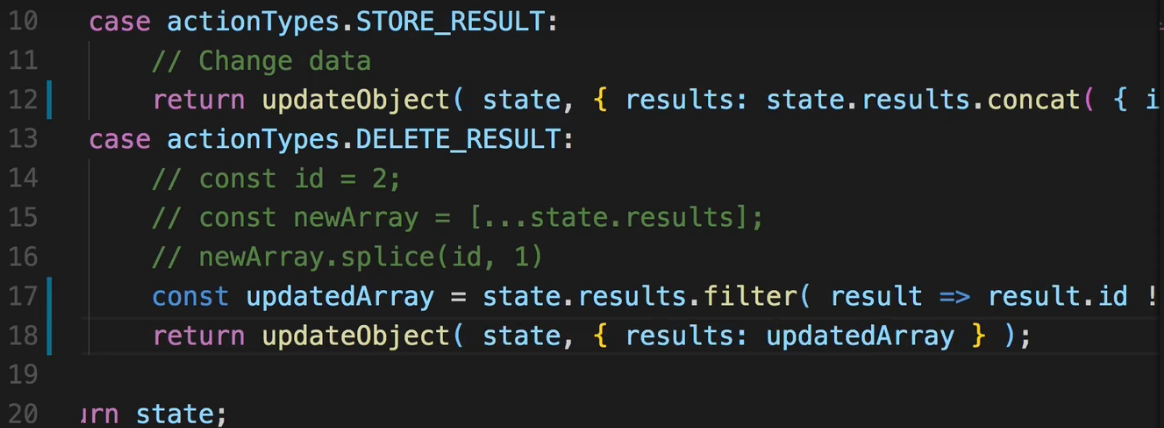
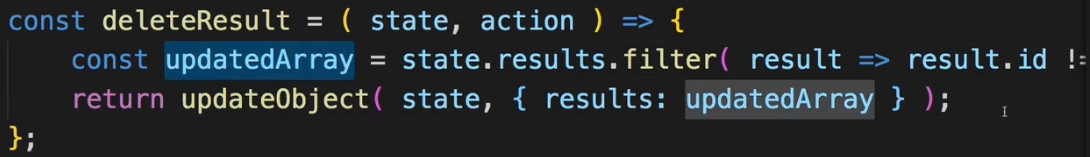
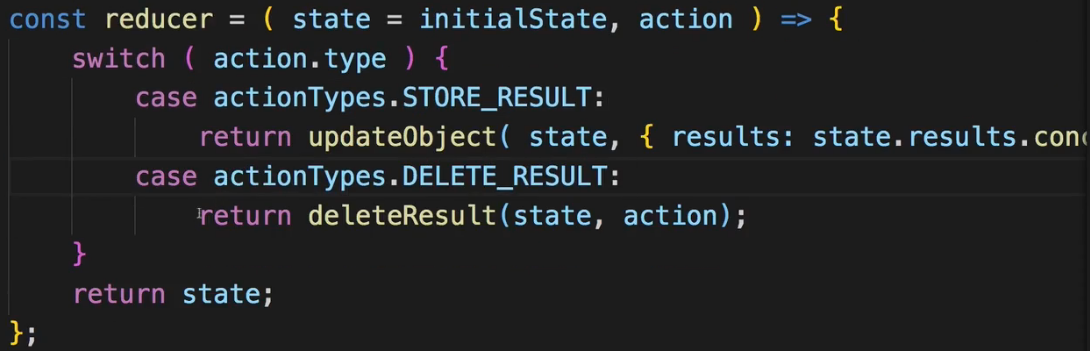
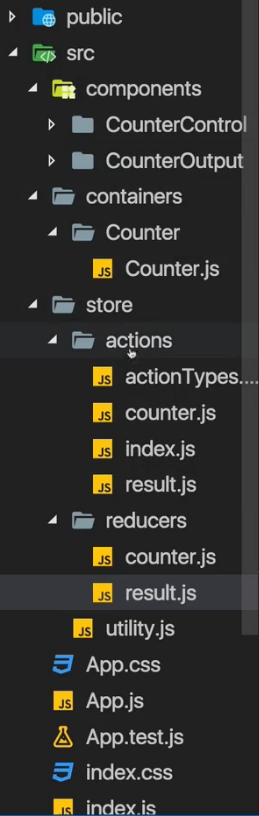
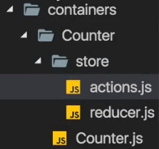
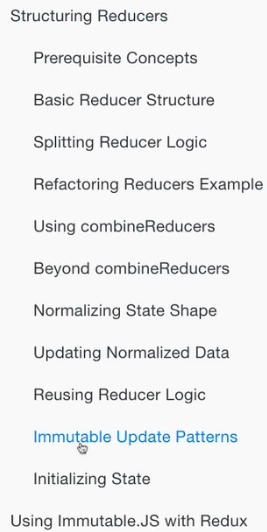
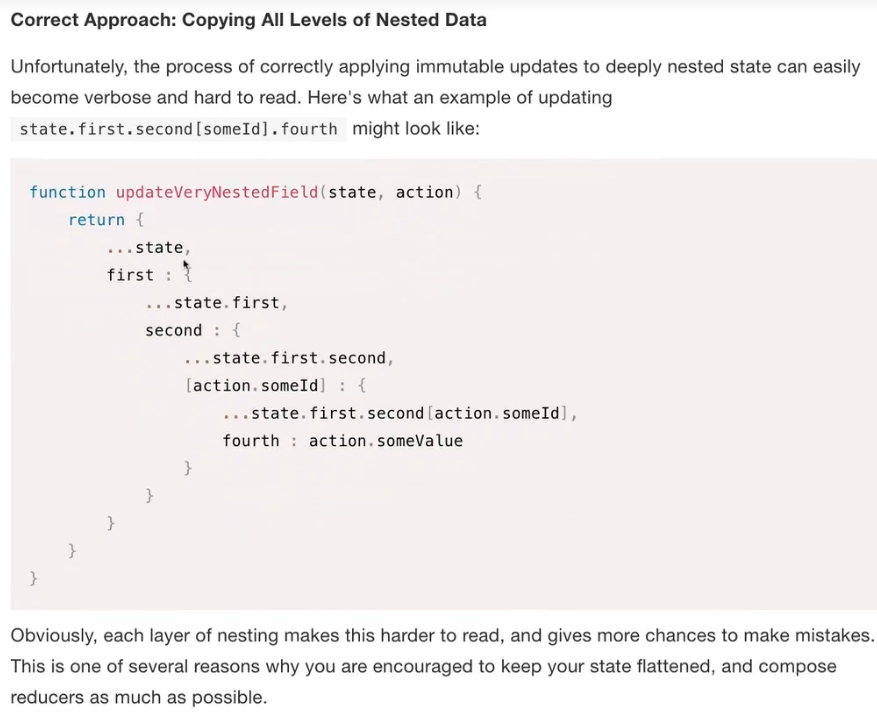
**Section 16 done: 16/16 Redux Advanced**  
**Adding Middleware**  
\* Before we dive into running asynchronous code, there’s one super important advanced concept we have when working with Redux, not necessarily connected to React.  
\* You may add MIDDLEWARE to it, right between your ACTION being dispatched and it reaching the REDUCER.  
\* MIDDLEWARE is a term used for functions - or code in general - you hook into a process which then gets executed as part of that process without stopping it.  
  
=> We can do something with that ACTION before it reaches the REDUCER.  
=> That will also before important later when we want to execute asynchronous code.  
\* In `index.js` where we create the STORE. It’s at this point of time that we can also add MIDDLEWARE to the project.  
\* I’ll create my own MIDDLEWARE here, later we’ll add MIDDLEWARE provided by other providers.  
\* It takes `store` as an argument because we will soon use a specific method provided by Redux to connect our own MIDDLEWARE to the STORE and this method provided by Redux will eventually execute our MIDDLEWARE function and give us the STORE.  
=> We return a function in our MIDDLEWARE function.  
=> That function also returns a function which will receive the ACTION you dispatched as an input.  
  
=> We need to import `**applyMiddleware**` from redux.  
=> Now this allows us to pass a 2nd argument to the .createStore() and the 2nd argument here can be a so called **enhancer**. Now this enhancer is nothing else then a MIDDLEWARE for example.  
  
\* And actually you can pass a list of MIDDLEWAREs here to applyMiddleware(), they will be executed in order then. (example: applyMiddleware(first, second, third)).  
  
\* Now of course the MIDDLEWARE can already be nice to do exactly that - log your state and see very well what’s going on.  
\* Now a more useful use case for MIDDLEWARE is - which we’ll se later when we actually handle asynchronous code - but first, I wanna stick to this idea of getting some insight into this state, it would be nice for debugging if we could always look into the STORE. So if we had some logging but more than that, that even if we just didn’t dispatch anything, we could still look into the current STORE.  
**Using the Redux Devtools**  
\* For the Redux Devtools to see our STORE, we simply need to pass a special **enhancer** to our createStore() method.  
  
\* Since we’re using MIDDLEWARE, we have to use the Advanced store setup:  
  
=> This essentially is a variable which is injected by the Chrome Extension into our JavaScript at run-time so it will be available.  
=> If that variable can’t be found, we will fall back to a default `compose` function provided by Redux.  
=> Now `compose` is a little bit similar to `combineReducers`, because `compose` allows us to combine enhancers.  
=> `applyMiddleware` is only for MIDDLEWAREs, if we had other enhancers like the store devtools, we need to use `compose` to compose a set of enhancers with both the devtools features and our MIDDLEWARE.   
|| compose;  
  
\* Now the devtools work.  
  
\* 1 important feature of Redux DevTools is “time traveling” - you can hover over an ACTION and click “jump”. And now you’re back to that point of time when that ACTION happened. You can always go back by clicking “jump” on the last ACTION. You can also click “skip” to update your state as if you never dispatched this ACTION.  
\* This is an extremely nice feature which gives you a lot of debugging possibilities especially in more complex apps. This is great to make sure that only ACTIONs are dispatched which we expect to be dispatched and to find out why the state is currently in the state it is in.  
**Executing Asynchronous Code - Introduction**  
\* We don’t have any asynchronous stuff happening in this demo app but we can easily simulate one - let’s say that when storing a new result that we also store this on a server - now we won’t reach out ot a server here because it doesn’t matter but we can simply use **.setTimeout()** to simulate that this simply takes a short period of time. And we want to update the store after this happens for example. The same for deleting a result.  
=> So we want to create Asynchronous ACTIONs.  
=> The REDUCER function has to run synchronously, you cannot add .setTimeout().  
=> So typically what you would do is return a PROMISE which calls RESOLVE inside of .setTimeout() after a certain period of time but you REDUCER just doesn’t expect to get a PROMISE.  
=> So there is just no way you can execute asynchronous code in the REDUCER.  
=> There is a different way though.  
=> We can execute asynchronous code with the help of so called **ACTION CREATORs**.  
**Introducing Action Creators**  
\* Let’s create an `actions` folder inside of our `store` folder and move our `actions.js` in there. I’m moving it there because I’ll also add multiple ACTION files over the next lecture but for now let’s stick to this one.  
  
\* In there we export all our ACTION IDENTIFIERS and that is working fine.  
=> Now I want to introduce a new way of creating ACTIONS - so called ACTION CREATORs.  
=> It’s just a function which returns/creates an ACTION and hence the name.  
=> We right now create our ACTIONs hard-coded in our code simply where we need them.  
=> In the Counter.js component here on mapDispatchToProps the ACTIONs these are just these JavaScript objects, we create them here. We create them by simply writing the code for the object into our dispatch() function.  
  
=> An ACTION CREATOR would return such an object.   
\* And you’ll see the benefit when we talk about asynchronous code.  
\* In `actions.js` after exporting the action identifiers:  
\* Let’s create an ACTION CREATOR for Synchronous code.   
=> The convention is to use the same name as the identifier but in camelCase.  
=> This function receives any PAYLOADS you want to pass with that ACTION.  
  
=> This is the bare minimum ACTION we can create - just with the `type`.  
=> We can use this ACTION CREATOR in our Counter.js file.  
  
\* Keep in mind that this is a function.  
  
=> **Let’s pass the function and also EXECUTE it because once I execute `increment()`, and this will happen when this prop is basically executed, this will give me an ACTION because it’s an ACTION CREATOR. So we just get an ACTION**.  
\* Now let’s create ACTION CREATORs for all the ACTIONs we have.  
**Action Creators & Async Code**  
\* For ADD we need a PAYLOAD.  
\* How do we get that PAYLOAD into our ACTION CREATOR?  
=> Well keep in mind we execute it here in our container so this is where we probably want to pass any PAYLOAD as arguments to the ACTION CREATOR.  
 

   
  
  
\* I still dispatch the same ACTIONs as before, this didn’t change.  
\* This far only Synchronous code runs in them.  
\* Q: Should we use ACTION CREATORs for Synchronous code or not?   
=> The Answer is: it’s up to you. It’s a clean way of creating your ACTIONs, you have everything about ACTIONs in 1 file, you don’t have to create the ACTION objects anywhere else, this is a tiny clean up which can be considered good so nothing speaks against using ACTION CREATORs for Synchronous code too but as I said, it will be extremely useful for Asynchronous code.  
\* As I said, ACTION CREATORs are useful for handling Asynchronous code so let’s dive into that next.  
**Handling Asynchronous Code**  
\* To handle Asynchronous code we need to add a special MIDDLEWARE to our redux project.  
=> A third-party library we can add called **redux-thunk**.  
\* Generally, this is a library which adds a MIDDLEWARE to your project which allows your ACTION CREATORs to not return the ACTION itself but return a function which will eventually dispatch an ACTION.  
=> With this little trick - not returning the ACTION itself but a function which will then dispatch one - we can run Asynchronous code. Because the -eventually dispatched one part- is the part which may run Asynchronously.  
**npm install --save redux-thunk**  
  
=> We then just add it with `applyMiddleware`.  
\* They actually have a default export so we don’t need {}, we can give it any name we want.  
 \* this package essentially just exports the MIDDLEWARE.  
=> This `thunk` already is a MIDDLEWARE behind the scenes.  
  
\* Now let’s say in `storeResult` ACTION CREATOR we want to execute .setTimeout() and after 2s we want to store the result. To simulate that we prior to this ACTION we reached out to a server to store it there and only update our state after this was successful.  
  
\* In this current set up it wouldn’t work.  
\* But with the `thunk` MIDDLEWARE we can change this return statement here which will return a function and that’s important.  
  
\* Now we get `dispatch` in here due to redux-thunk. I said that MIDDLEWARE runs between the dispatching of an ACTION and the point of time the ACTION reaches the REDUCER.  
\* Now the thing we do here is we still dispatch an ACTION but then redux-thunk - the MIDDLEWARE - comes in, steps in, has access to the ACTION there, basically block the old ACTION we could say and dispatches it again in the future. Now the new ACTION will reach the REDUCER but inbetween redux-thunk is able to wait because it can dispatch an ACTION whenever it wants. This is the Asynchronous part and that is exactly allowing us to execute some Asynchronous code inside of this function.  
=> So the code inside of this dispatch function here is executed and now inside of .setTimeout() - inside of this function passed to .setTimeout(), we can execute dispatch() to now dispatch whichever ACTION we want to dispatch.  
\* Now of course we would create an infinite loop if we again dispatch(storeResult) - the ACTION CREATOR - here:  
  
=> So what we typically do is we create Asynchronous ACTION CREATORs which in the end dispatch() ACTIONs created by Synchronous ACTION CREATORs.

=> **Synchronous ACTION CREATOR**  
  
=> **Asynchronous ACTION CREATOR** => which eventually calls the **Synchronous ACTION CREATOR**  
  
\* Now before we see this in action, there’s 1 thing we have to keep in mind - here in storeResult where we return this function, which will get executed by redux-thunk and where we have .setTimeout(), where we then dispatch() the ACTION which should run synchronously and update the STORE - **we need to EXECUTE saveResult() - the Synchronous ACTION CREATOR - as a function of course and pass `res` on**.  
\* Now if we click “store result” button, nothing happens in the Redux DevTools, then after 2 seconds we see the ACTION. That’s because only Synchronous ACTIONs may edit the STORE. The other ACTION CREATORs like `storeResult` which run some Asynchronous code are only possible due to redux-thunk and are caught inbetween, they never make it to the REDUCER. We only use them as a utility step inbetween to run our Asynchronous code, which happens to be required to run on a lot of ACTIONs, and then dispatch the Synchronous ACTION to change the state in the STORE - once we are certain that we know what we want to do there, so once our Asynchronous code finished.  
\* In the console.log() we got more output because the `logger` MIDDLEWARE we created logs everything which reaches the ACTION funnel and that includes our function which is returned by the Asynchronous ACTION. We never edit the state here though because that gets blocked by redux-thunk.  
**Restructuring Actions**  
\* Now our `actions.js` file holds the ACTION IDENTIFIERS and also all these ACTION CREATORs which use these identifiers.  
\* Obviously, our file here in the demo project isn’t that big, we can of course stick to 1 file but we also split up the REDUCERs into `counter.js` and `result.js` and you typically also do that for ACTIONS as your application grows and you’re going to see that in the BurgerBuilder project.  
 \* It should be `actionTypes` not `actionsTypes`  
=> Let’s create the `actions.js` and `result.js` just like for the REDUCERs and I’ll rename the `actions.js` to `actionTypes.js` because here I now only want to export these unique identifiers. And theoretically you could of course also split that up into multiple files.  
\* Now in `counter.js` and `result.js` we need to adjust the reference to the actionTypes.  
  
\* Now I also want to have 1 file exporting all my ACTION CREATORs.  
=> So I’ll create an `index.js` file and in there I’ll actually use a syntax you might have not seen before, I’ll just export something from a file. I can do that. So I don’t even import it in this file, I just have 1 file - the `index.js` file which groups all exports from separate files so that in the end I can always point to that file to import something from any of the files I’ll point to in that file here.  
   
\* And this really is just an advanced feature or some advanced set up of the project to handle bigger projects with lots and lots of ACTIONs and ACTION CREATORs. It’s overkill for this demo project but it will make more sense later when we reach our BurgerBuilder application.  
\* We don’t export `saveResult` because I’ll never need it in any other file so I won’t include it into my grouping.  
\* Now this allows me to later just import from `index.js` and actually import from any of these 2 files which I grouped in this `index.js` file.  
\* Now that’s just a tiny improvement to make our files even a bit leaner.  
\* Now I need to adjust my REDUCER files where I do import these `actionTypes`.  
  
=> This in both `counter.js` and `result.js` REDUCER files.  
\* And in our `Counter.js` file we point to the ACTION CREATORS:  
  
**Where to Put Data Transforming Logic?**  
\* Now I want to dive into what we actually put into the ACTION CREATORs.  
\* It’s obviously easy for Asynchronous code, the only place where we can execute Asynchronous code is in our ACTION CREATOR, it’s what redux-thunk is made for and it’s the common and best practice pattern. Send you HTTP Request there instead of .setTimeout() and once the response is there, store it in your STORE.  
\* However, you can of course put much more logic into your ACTION CREATORs.  
\* Think about saveResult, we save our result there we get it as an argument `res` and we simply return an ACTION where we pass it on as a PAYLOAD.  
  
=> Now this is a very dry ACTION CREATOR, it doesn’t do anything else but just return an object with the unchanged result.  
=> Now obviously what we could do is: we could alter anything we want here, like this:  
  
=> And we might have transformations which make more sense - maybe we want to update some ID, we want to add a user name, and then we pass on our updated result.  
\* Now we have logic in our ACTION CREATOR and this might be valid logic instead of some nonsense operation like this one.  
\* The thing is - you could of course also execute the same logic if you need to transform the data before storing it in the state, which is perfectly fine, which might happen - you can execute that same logic of course in your REDUCER.  
  
\* You’ll often have cases where you really want to change something before you store it in the state, you not always get a value you just want to pass on.  
=> Where should you then change it?  
=> In the REDUCER?  
=> In the ACTION CREATOR?  
   
=> Both works.  
=> What’s better?  
=> In the end the question comes down to: Where to put the Logic?  
=> ACTION CREATORs are great for running Asynchronous code when you dispatch an ACTION.  
=> REDUCERs only are able to run Synchronous code and are PURE = input in, updated state out.  
=> REDUCERs however - keep that in mind - are meant to be the place where you update the state. This is 1 core Redux concept.  
=> ACTION CREATORs aren’t a core Redux concept, a core concept are ACTIONs - these JavaScript objects with a TYPE and a PAYLOAD.  
=> So the REDUCER is a core concept and the whole idea behind Redux is that the REDUCER is the only thing which updates the state.  
=> ACTION CREATORs should’nt prepare the state too much for that reason because it should be the REDUCER which does the update.  
=> But there of course also is difference between updating the state - (which essentially just means returning a new object which makes up our state) - and changing the data which goes into the state.  
=> Still, you can find arguments for both directions, I lean towards putting the Logic into the REDUCER and not too much Logic into the ACTION CREATOR. Asynchronous code has to go there. But once you got back the data from the server you might need to reach out, you can of course transform it in the ACTION CREATOR, and you should do that to a certain extend, but once you got data that is relatively clean, you should hand it off to the REDUCER and if you then still need to manipulate it, for example by taking it \*2 or anything like that, in my opinion, that should go into the REDUCER.  
\* In the end it’s your decision.  
\* If you choose 1 approach, stick to it though, don’t change it, don’t put a lot of Logic into 1 ACTION CREATOR just to then have a lot of Logic in another REDUCER, be consistent and decide where do you want to transform and prepare your data, I recommend the REDUCER.  
**Using Action Creators and Get State**  
\* If you chose the ACTION CREATOR, here’s another UTILITY METHOD you might wanna know when working with thunks - so Asynchronous code handled by redux-thunk.  
\* For example our storeResult here.  
=> Actually redux-thunk can pass us additional argument `**getState**`. That is a method we can execute to get the current state.  
=> Sometimes in your Asynchronous code you need to be able to reach out to the state prior to the dispatch() ACTION. Let’s say you want to save some data for a given user, and you have the ID of that user stored in your Redux state, you can then get it with getState().  
  
=> For example here we can get the old `counter` with getState().counter.  
=> getState() will give us the complete state which has the `counter` property. Or to be precise - since our REDUCERs make up the state, and we have 2 of them, we can get the `counter` just as in the Counter.js container in mapStateToProps we have to acccess .ctr.counter.  
\* Now this is kind of related to what I said in the last lecture - if you need it, it’s a nice UTILITY FUNCTION, don’t overuse it though. I tried to write my ACTION CREATORs and REDUCERs in a way that I don’t have to use getState(). Instead, you can pass all the data you need in your Asynchronous ACTION CREATOR - like the user ID - into it by accepting it as an ARGUMENT. That of course means that you need to have access to the data you need in your ACTION CREATOR - in the container where you actually dispatch() the ACTION leading to the ACTION CREATOR. So here:  
  
=> So we need to have access to the user ID in this Counter.js container.  
\* But that is something we might be able to expect and we should build our app accordingly. If you just can’t do that or don’t want to, you have getState as a fallback, you can use it, just don’t overuse it. Don’t put too much Logic in there, at least that is the route I take in my bigger React applications.  
**Using Utility Functions**  
\* Now I want to dive into advanced REDUCER setups.  
\* Right now we have 2 REDUCERs - it’s a bit advanced because we already split it up and used `combineReducers` but each REDUCER still has a relatively long `switch` statement.  
\* So cleaning up that REDUCER files a bit more can be an idea - it’s not a must though. It’s a good practice I’d say but you don’t have to do it.  
\* I want to show you some ways of cleaning up this REDUCER step by step though to end up with a very clean REDUCER.  
=> The first thing we can do is related to immutability.  
=> We constantaly update an object here in the end, whenever we return a new state, what we do is just return an updated object. We return an object which copies the properties of the old state and then replaces 1 of the properties.  
  
=> Now obviously we can absolutely do it like this.  
=> But we could create a UTILITY FUNCTIONs for that. And the same for updating arrays.  
=> This would then allow us to call this UTILITY FUNCTION and save some code here:  
  
=> In the `store` folder I create a new file `utility.js`, you could store it in the `reducers` but I don’t want it to look like a REDUCER.  
   
\* Now in `counter.js` REDUCER:  
   
  
\* Now we’re using this UTILITY FUNCTION for all the ACTIONS in REDUCERs.  
  
\* In `result.js` REDUCER we’re also updating an array in the DELETE\_RESULT:  
  
\* I won’t write a UTILITY FUNCTION for that since updating an array depends highly on what we do with an array => adding an element works with .concat(), removing an element works with .filter() for example, so therefore it’s not as easy as with an object and the UTILITY FUNCTION doesn’t make that much sense for that reason.   
\* But we still update an object in here:  
  
\* Since we always return a new state in the end.  
**A Leaner Switch Case Statement**  
\* By using our updateObject() UTILITY FUNCTION we made our REDUCERs leaner by outsourcing a common step we repeated all over the place, into a UTILITY FUNCTION.  
  
\* Now our REDUCER cases actually are pretty lean with that.  
\* In this `counter.js` REDUCER they can’t get much leaner.  
\* In the `result.js` REDUCER we at least have 1 of the cases with the extra step where we update the array.  
  
=> Some developers like to have very lean cases where you essentially have “case: 1 function” you call. Obviously we still need to run our Logic but they like to outsource this into a function so that the `switch` statement is very lean.  
=> This has 1 major advantage: if a `switch` case statement looks like in the `counter.js` REDUCER, we can quickly see what happens in each REDUCER or at least where we handle something.  
=> For the `result.js` REDUCER file we could outsource the extra Logic into its own function, that function goes into the same file. The convention should be to use the ACTION TYPE as the name, though in camelCase.  
\* So I need to have the `state` and the `action` as arguments - so essentially the same idea I get in the REDUCER.   
  
=> Basically doing the same as we did before in the `switch` case statement, and that’s exactly the idea behind these helper functions.  
  
\* You could also put the case and return all in 1 line if you wanted to optionally.  
\* This improvement makes it easy to see which cases you’re handling in a given REDUCER and that can be the advantage of this approach.  
**An Alternative Folder Structure**  
\* Now we have leaner REDUCER files. Generally we split our ACTIONs and REDUCERs across a lot of files and create a lot of BOILER PLATE code here.  
\* For a small project as this one, this is clearly an overkill but also it allows me to show you what you can do and that will become important in bigger projects. We’ll already see some advantages in the BurgerBuilder app.  
\* I wanted to highlight these possible improvement or restructurings to allow you to write as lean and understandable REDUCERs, ACTIONs, and React applications in general, as possible.  
\* I want to quickly dive into an alternative folder structure.  
  
\* Now the folder structure really is something you can find thousand of different approaches.  
\* The approach I chose in this course is the one you’ll probably see the most often in React projects, it’s quite good, with the `components` and `containers` differentiation and then the `store`.  
\* Now as your `store` grows, you might add more and more ACTION and REDUCER files. And you’ll notice that some of your REDUCER files refer to some container and other REDUCER files to another container.  
=> For that reason in an alternative project structure you could create a `store` sub-folder in every container and then only add the REDUCER and the ACTIONs for that given container, to that `store`. And if you had more containers, each container folder would have a `store` folder, you wouldn’t have that global `store` folder, and you would have a lot of ACTIONs and REDUCERs for each container, which you then still combine together in your `index.js` file though.  
  
\* So this is an alternative, you can also mix that with the global `store` for some global or shared operations you might need everywhere, that really is something you can fine-tune and where you can read a lot of articles with a lot of opinions. Find the approach you like.  
\* For many projects, the approach shown here with a central `store` folder, should be fine but be aware that you’re free to deviate from that when needed  
**Diving Much Deeper**  
\* I’m now on: <https://redux.js.org/> the official page of the Redux package.  
\* And keep in mind that Redux is a standalone package, it’s not a part of React.  
\* But of course it’s commonly used with React which is why you find a lot of Redux + React resources on that official Redux page too.  
\* Of course you’re taking the course to learn stuff by viewing a video.  
\* Redux, however, has so many use cases and possible adjustments you can look into that I strongly recommend having a look at this page, if you feel like your current approach has you stuck or your REDUCER functions are exploding.  
\* You can learn more about possible alternative practice and best practices here.  
\* Now most of the content covered on this page has been covered in the Redux Basics and this Redux Advanced module though.  
=> So no need to dive right into it right now and go through all of that.  
=> Just see it as a fallback page you can go at and look at if something is unclear or there is some use case where you’re not sure how to handle it.  
\* 1 especially useful resource I want to point you to though is the `**Immutable Update Patterns**` article here **under `Recipes`**.  
  
=> You learn more about how you update nested objects and arrays, immutably.  
=> This can be tricky since you have to go down to all nested levels.  
=> If you have an object with an object with an object inside of it and you want to change something in the deeply nested object, you have to clone all 3 objects first, as shown here, before you can actually update some immutably.  
  
\* And you can see some patterns and examples in this article.  
\* That might be a great help to make sure you’re updating everything in the way you want to update it.  
**Wrap Up**  
\* I emphasized that there are alternatives.  
\* And I showed you some best practices, good practices and ways to optimize your REDUCERs and ACTIONs.  
\* You learned how to run Asynchronous code with ACTION CREATORs, how to use the redux-thunk MIDDLEWARE to do that.  
\* You learned how to combine REDUCERs and how to use the Redux DevTools to get insights into your STORE and even travel back in time if you want to.  
\* These are all useful tools and Redux is an extremely powerful library you’ll see in a lot of React applications.  
**Middleware**: <http://redux.js.org/docs/advanced/Middleware.html>  
**redux-thunk** **package**: <https://github.com/gaearon/redux-thunk>  
**Async** **Actions**: <http://redux.js.org/docs/advanced/AsyncActions.html>