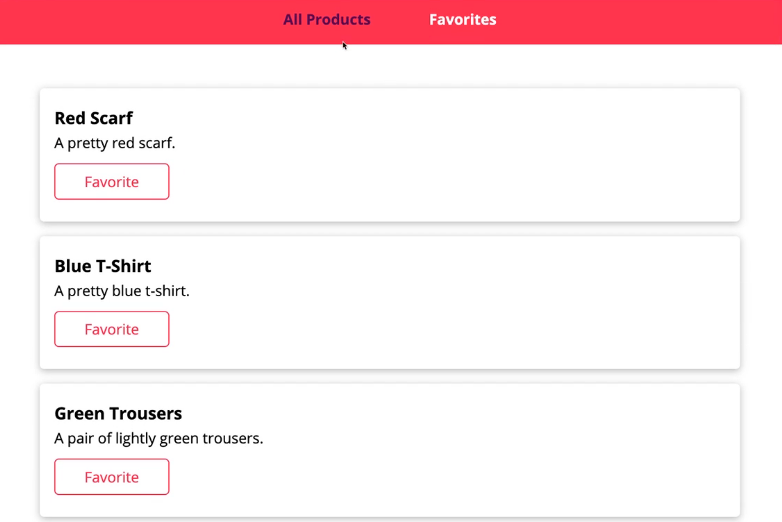
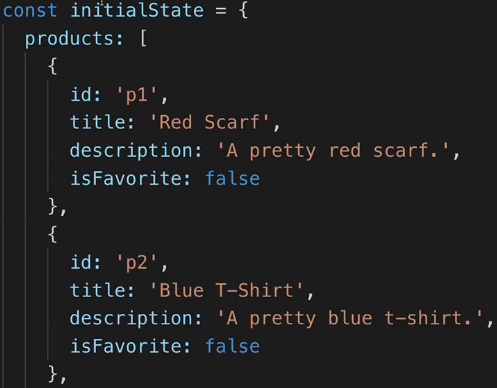
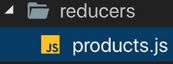
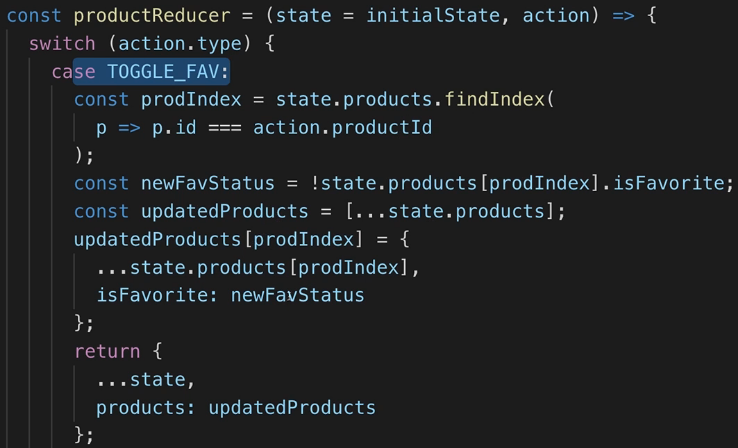
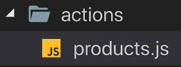
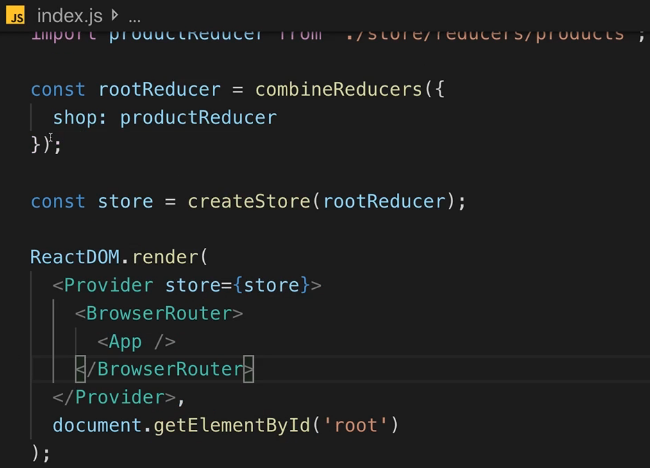
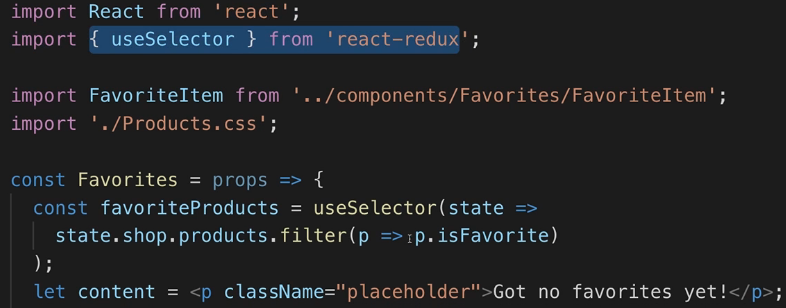
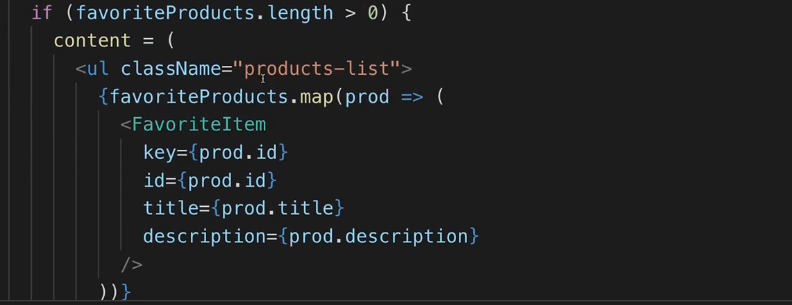
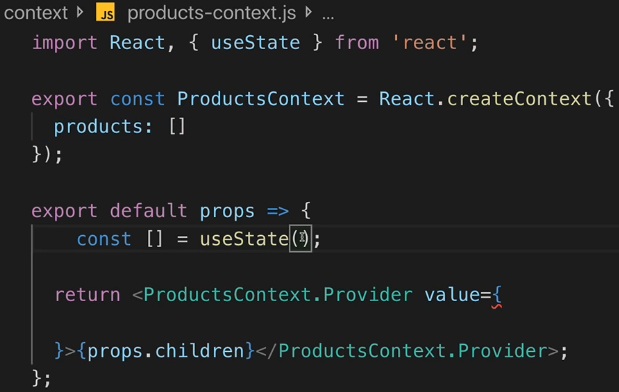
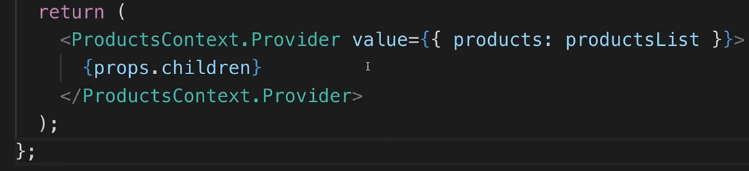
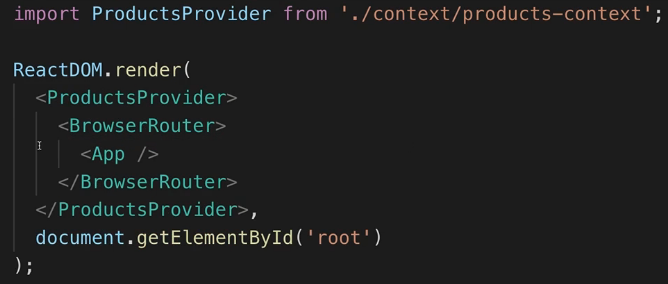
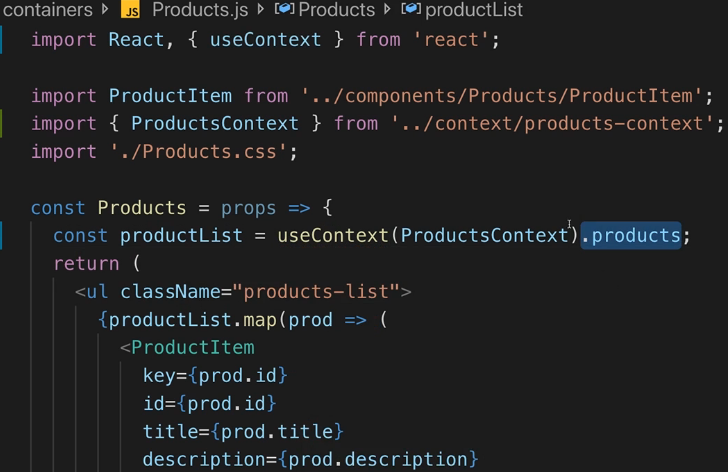
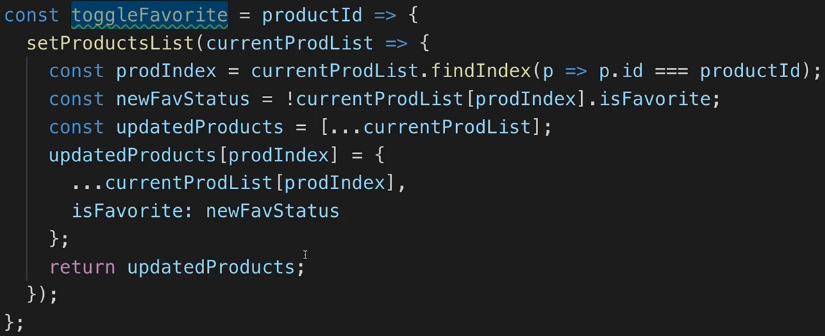
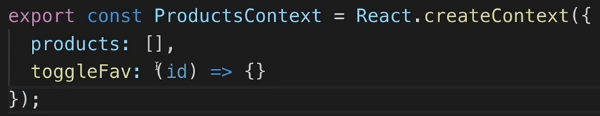
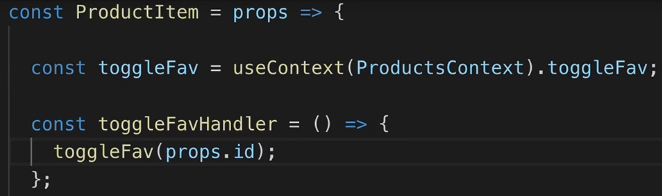
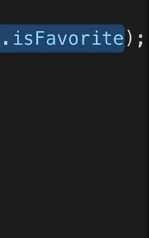
**Module Introduction**  
\* **Replacing Redux with Context API and/or Hooks?**  
\* As it turns out, we can actually replace Redux as our global state management system with one of these tools.  
\* **Let’s explore why we might want to do this and how we can do this**.  
\* **And also why/how we should not do it in a certain way**.

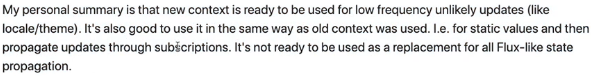
**Starting Project & Why You Would Replace Redux**  
  
\* A very simple application where we got a list of products and we got a “Favorites” section and we can basically switch between these pages and we can mark something as “Favorite”.  
\* **Right now it uses Redux to manage this “Favorite” status**.  
=> We need state in 2 different pages and that would be harder to do with props only and I don’t want to pass all the state through multiple layers of components just so that I have it everywhere where I need it and Redux is really convenient for such cases, it’s a great state management tool when you have some global state.  
\* I use the traditional Redux approach I showed you in the course.  
\* products.js REDUCER:  
   
\* There’s some initial state which initializes my products.  
  
\* products.js in actions.  
   
\* That is connected to the Products.js.  
  
=> With the help of the useSelector() Hook added by react-redux version 7+ which I covered at the end of the module where we transformed our BurgerBuilder to use Hooks, this Hook is simply used to select a slice of our state - the shop slice which we create in index.js:  
  
\* And we select the products list from that slice and output it here.  
  
\* In Favorites.js I also select something with useSelector() from our store and I select all products that are a favorite.  
  
  
\* And lastly in the ProductItem.js I use the useDispatch() Hook provided by react-redux so that when we click that “Favorite” button, we dispatch this toggleFav() action which is defined in the actions folder of our store folder.  
  
  
\* So that’s how our app works.  
\* **The first important thing I really want to emphasize is that there’s nothing wrong with that approach**.  
=> You don’t need to switch away from Redux at all.   
\* But maybe you want to - and why would?  
=> Well maybe you want to stay in the React-only world so that you don’t need to learn Redux. Arguably it’s not that hard to learn Redux but still maybe you want to use only the tools React gives you.  
=> **Maybe you also simply don’t want to add that extra redux and react-redux library to your project hence you would end up with a smaller bundle** because if you don’t have these extra libraries included, obviously you ship less code when you deploy your application and that could be a good argument because you can certainly shrink your bundle a bit, though I will also say **if you have a really large application, it might not matter that much if it’s included or not**. Yet, that could be a reason for why you don’t want to use Redux.  
=> **Or you simply want to explore how you can manage your state globally without passing everything through props but still without Redux**.  
\* I’m going to show you 2 different approaches of managing your state with React-only tools.  
\* One of the approaches is a good one.  
\* The other one is not that great - at least nmot for all use cases.  
\* I’ll show you both and I’ll explain when to use which.

**Alternative: Using the Context API**  
\* This approach is sometimes the right choice but not always.  
\* **Context API**  
  
\* In this case you could argue that your entire context value is just an array but in case you had other fields too, you would probably have an object here.  
\* With that we’re creating React Context object here.

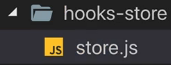
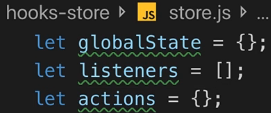
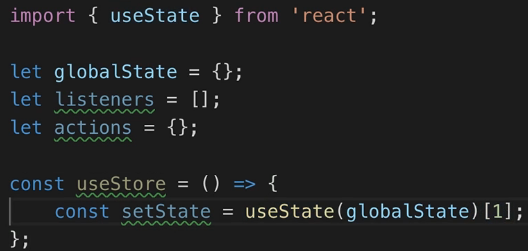
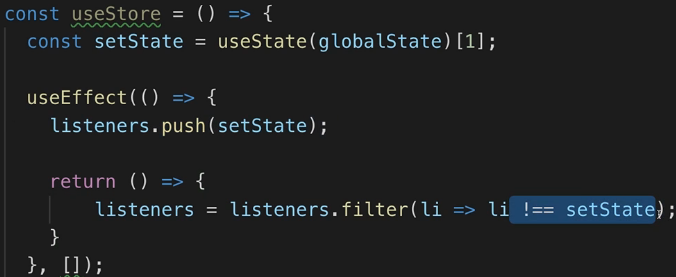
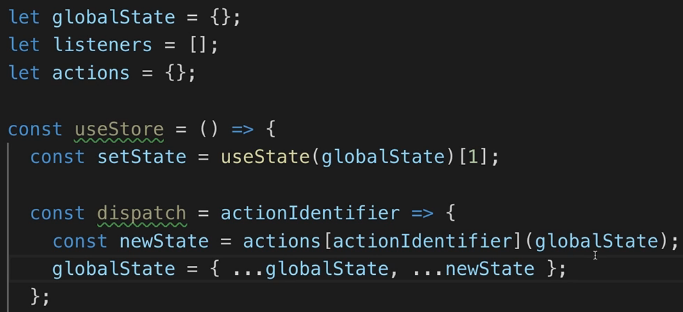
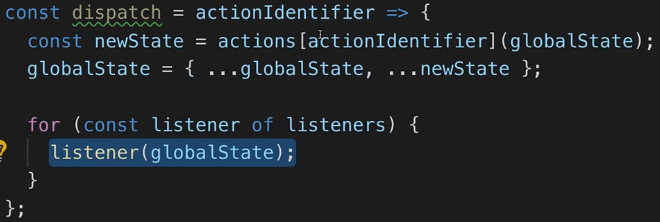
\* Now I want to export this Context object.  
\* But I also want to export a component as a default which gets some props and returns some JSX, so I’m exporting just a normal Functional component here.  
\* Important - the value of this Context should be a value which I manage in this state.  
\* Since that should be able to change, I’m going to use useState() Hook. (theoretically that could be a class based component here too).  
\* My state I manage here in the end will be that array of products, so actually I’ll grab that array and remove it from the Context and simply initialize an empty array to get better autocompleetion and IDE support but the value here doesn’t matter because we’ll now manage the products here in the state.  
  
\* Let’s add it to the state:  
  
  
\* So now whenever we update the state and therefore this component rebuilds, our Provider will get a new value and every child that listens to our Provider will be able to get that new value.  
\* Now with that prepared, we can use our Provider.  
\* Instead of the react-redux Provider.  
  
\* You could delete the entire store folder now.  
\* **Now in any child component here, I can kind of listen to that Context**.  
\* In the child, we import the ProductsContext object itself.  
=> **We can use ProductsContext together with useContext() Hook**.  
  
\* That’s our list of products which is recreated whenever we change something in that list.  
\* Now we’re able to render all our products.  
\* We can’t use “Favorite” for now but **we can use our Context to distribute data to a different component because we’re now managing our products in the ProductsContext and we’re accessing our products in Products.js**.

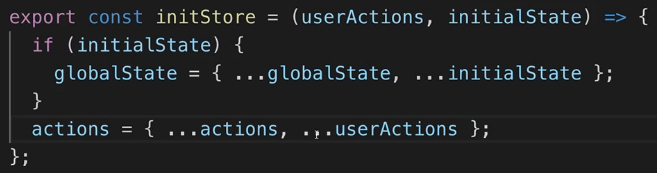
**Toggling Favorites with the Context API**  
\* In our Context file we can add a function where we expect to get the product ID - the ID of the product for which I want to toggle the “favorite” status.  
\* The logic here is pretty similar to what I did in the Reducer before.  
\* **I’m using the function form to be guaranteed that I’m working with the most recent state**.  
  
\* I’m adding this function and ID here just so that we get better IDE support and autocompletion.  
  

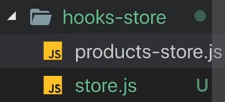
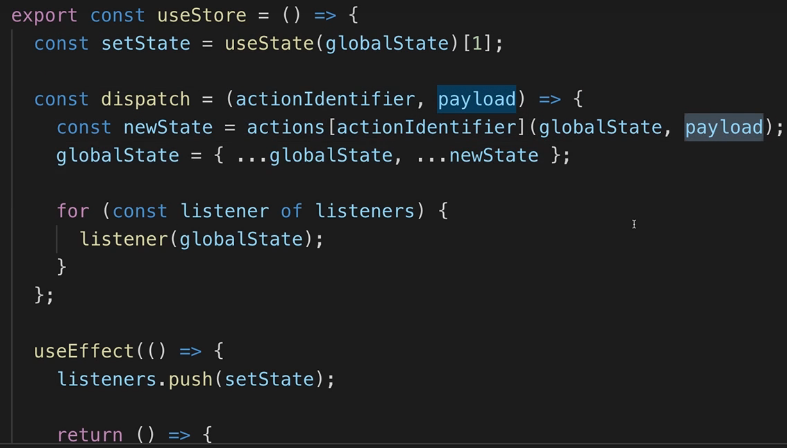

\* **Now everywhere where we tap into our Context, we can call this function and then this function will change our state and therefore emit a new state, a new value to all components that are listening**.  
  
\* Now using “Favorite” works again.  
\* Now let’s make the “Favorite Page” work again.  
  
\* Now this page works again.  
\* **So now we’re managing our state with the Context API**.   
\* **And on first look that looks amazing, we’re not using Redux and it wasn’t too hard**.  
\* **We basically now have our store in the Context file and we’re using React-only features to pass that data around and to change it from anywhere and to read it from anywhere**.  
\* **What’s the downside of this approach?**

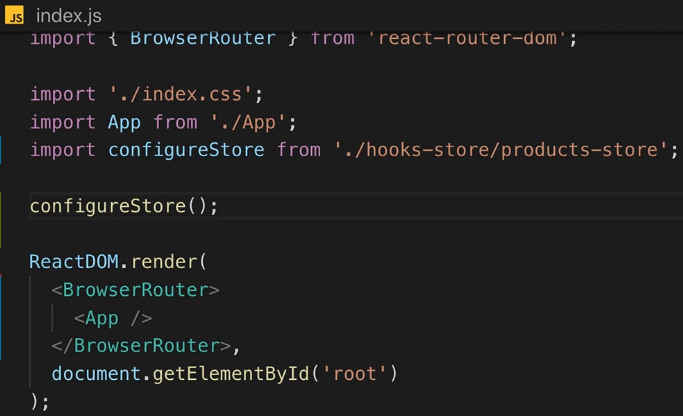
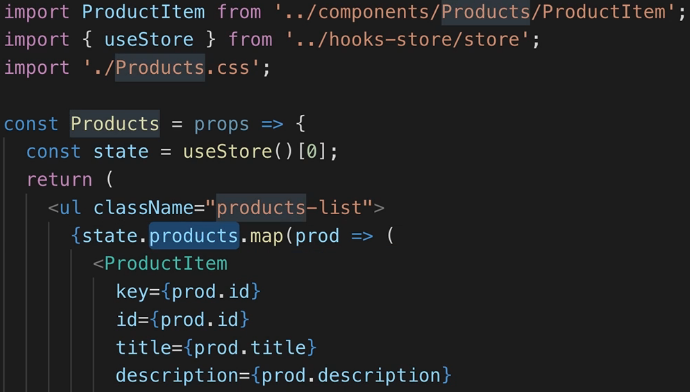
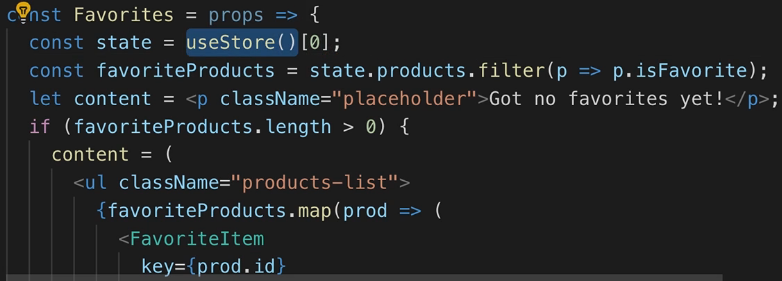
**Context API Summary (and why NOT to use it instead of Redux)**  
\* One huge downside is this comment by a React team member:  
  
=> **Great for low frequency updates but not for high frequency ones**.  
=> **If you have something which changes rarely - let’s say the user authentication status, which is the example I used in the React Hooks module, then it’s fine to be covered with the Context API because this will not change that often**.  
\* **The same would be the case if you had something like theming on a page - that also doesn’t change all the time**.  
\* Our “Favorite” status you could argue that it’s quite a high frequency update. It will very likely occur way more often than logging in or changing the theme of a page.  
\* And therefore I would argue that this is more on the high frequency side.  
\* Just like maybe managing a shopping cart.  
\* The Context API of course works in that case but regarding the performance it’s simply not optimized for that.  
\* **The way the React Context API works is such that whenever something changes in your Context, it has no way of cleverly figuring out which component that uses this Context really is concerned and which component is not**.  
\* **Which means that every component that uses useContext(), will re-render when you switch something in that Context, no matter if it’s directly affected or not**.

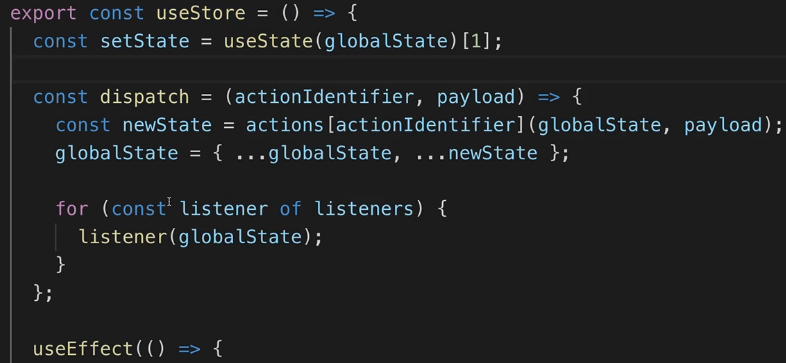
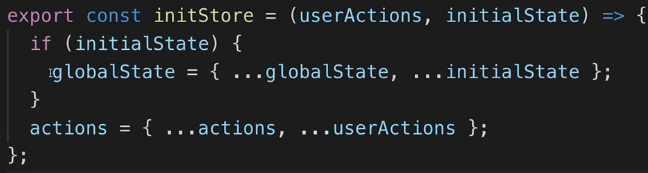
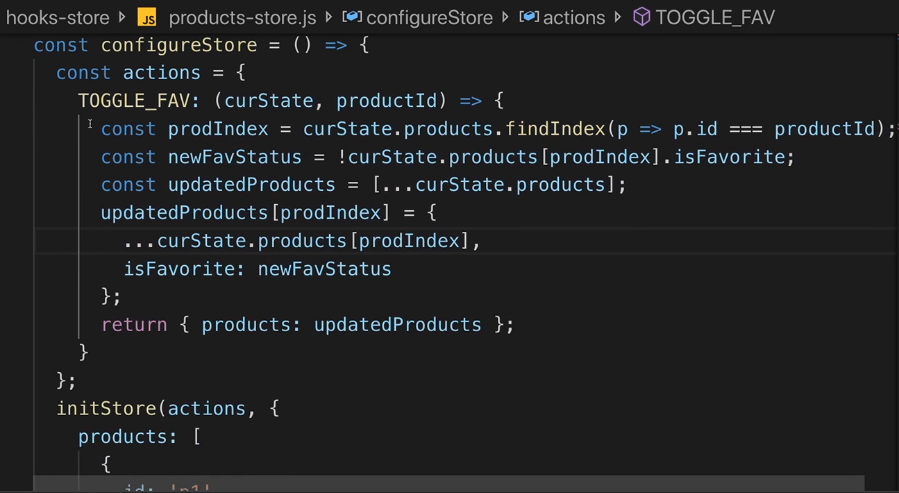
\* And in general, the React Context API is simply not optimized and not meant to be your global state management tool in your app.  
\* It’s meant for some state like authentication status or like the theme, buit not for all state because of these missing optimizations and also this missing intent behind the Context API.

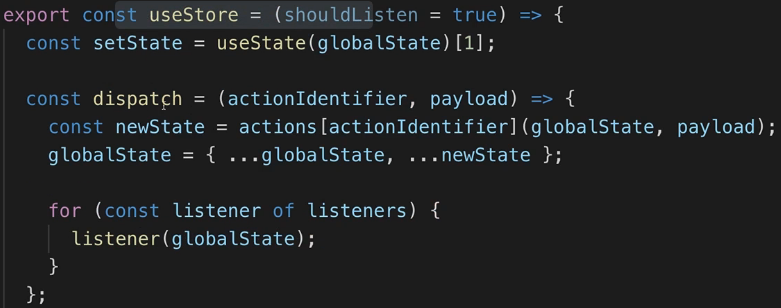
**Getting Started with a Custom Hook as a Store**  
\* Now let’s build our own global state management store and solution with just JavaScript, React and React Hooks.  
  
\* I’m deliberately using a very generic name here because we’ll be able to use the store for all kind of state, not just for our products.  
\* It could be an object.  
\* I also want to have a couple of places in the app where we can listen to changes in that state.  
\* Note that both `globalState` and `listeners` are normal variables, block-scoped so not globally available variables, they’re only available in that file and they’re not tied to any class or anything like that, they’re also not exported.  
\* We also need a couple of actions which can be dispatched.  
  
\* Now let’s create a custom Hook.  
\* **Let’s use the useState() because our goal here is that I have some mechanism that could lead to other components to re-render, and useState() has such a mechanism**.  
=> **useState() allows us to manage a state and whenever we update that state, any component that uses useState() will re-render. Custom Hook will cause a re-render when useState() inside of it triggers a re-render**.  
\* **I’m using the `globalState` which is DEFINED OUTSIDE OF THE HOOK, that’s important, it’s global, it’s not recreated when we call useStore(), it’s not created separately for every component that consumes the Custom Hook, instead, it will be created once - when this file is first imported somewhere and then any other file that imports from the same file will also use that same state.**  
\* So we’ll share data between all files that import from it.  
\* That’s not something we did before with Custom Hooks.  
\* **There the idea was the opposite - that we could share logic but not data**.  
=> **Now we’ll share logic AND data by managing the data outside of the Hook**.  
=> **Inside of the Hook, it would not be shared, it would be exclusive to each component, each component would get its own data**.  
  
\* I’m only interested in the updating function - setState value - you can name it whatever you want.  
\* I’m not interested in the current state snapshot.  
=> **I’m interested in the updating function because whenever this updating function is called, the component that uses my Custom Hook will re-render and we’ll need this later to re-render our components when something in our state changes**.  
\* I want to add this function to my `listeners` array because `listeners` should be an array full of functions which I can call to update all components that are using my Hook.  
  
=> **Every component which uses the Custom Hook will get its own setState() function which is then added to the shared `listeners` array**.  
\* That array will grow over time the more components we add.  
\* **If a component is unmounted, it would be great to get rid of its listener**.  
  
=> useEffect() with [] means this will only run for the component that uses the Custom Hook when that component mounts and we can also provide a Clean Up function here so that we remove the listener when the component unmounts because with the [] we’ll only run the useEffect() once and then run the clean up function when the component is removed.  
\* **Because this is a Closure, the value of setState is captured here for that component that’s using my Custom Hook and therefore will be the same when a component unmounts as it is when it mounts - even though setState is a function, a different object, it will be equal to the listener so to the same setState we registered here when the component mounted**.  
\* **We’re not entirely honest about our dependencies in useEffect() - we have setState as a dependency but that’s no problem because setState is coming from useState() and React guarantees that this never changes for a given component**.  
  
\* If we used the Array Destructuring syntax, we could omit this because then my linting tool was able to find out that this is the updating function which never changes - it’s not able to find out if we extract it like we do and therefore we have to add it.  
  
**Finishing the Store Hook**  
\* Now we need to be able to change our state and for that we should be able to kind of dispatch actions in our components - so the same idea as if we were using Redux.  
\* We’ll define the concrete actions in a different place, this here is still the overall abstract definition of our store management system.  
\* My idea is that we can later define concrete usages of our store with their own actions and therefore such actions will soon be registered here in the `actions` object.  
\* **`actions` should be an object where we have keys which match an identifier and where the value is a concrete function**.  
=> **The function should take our current global state and return a new state**.  
\* That’s essentially also what happened in Redux.  
  
\* And now we just need to inform all listeners about that state update.  
  
=> **This updates this React state because that’s where my listener comes from ultimately, with the new global state and hence React will re-render the component that is using my custom Hook here**.  
\* We’re still in a relatively abstract world but this was an important first step.  
\* **The Custom Hooks should also return something - an array of exactly 2 elements**.  
=> **That looks familiar - this is exactly what the built-in useReducer() function also returns**.  
  
\* Now let’s export our Hook:  
  
\* This Custom Hook is nice but it’s still very generic - for example right now we have no way to change these actions, we can call them but we can’t set them.

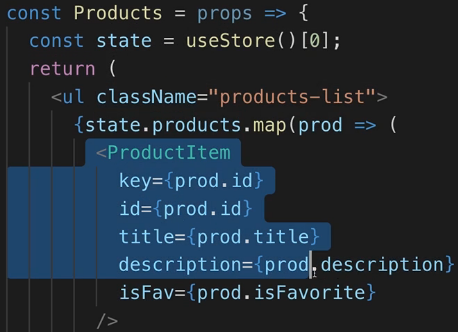
=> Let’s export another thing from this file:  
  
\* Our globalState is empty but maybe we want to set some initial state like the products we have.  
\* Keep in mind you’ll share that state, so just like in Redux you have multiple Reducers that make up one global store, so therefore in Redux you had to call combineReducers() to create 1 big store/state object.  
\* **Now we need some place where we can create a concrete products store based on this store setup**.

**Creating a Concrete Store**  
  
  
\* **Now let’s use the initiStore()**.  
\* Let’s export this configureStore function where all of this is happening.  
  
\* Now in our store.js file let’s make sure that when an action is dispatched and we forward the global state, we also can forward any value, any payload that is passed along with the action.  
  
\* The `productId` would be the payload that we need to pass along when we call dispatch.

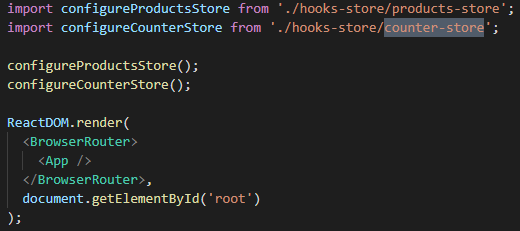
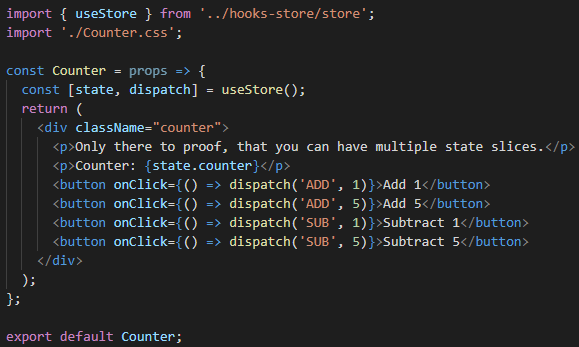
**Using the Custom Store**  
\* Let’s call the configureStore() to set up our store.  
  
\* **We don’t need to wrap this with a Provider component or anything like that**.  
=> This will make sure that we initialize our store since we import initStore in the configureStore(),the `globalState`, `listeners` and `actions` will all be set up and they will be initialized with the values we’re passing in our products-store.js file.  
\* If we had multiple stores, we would simply call configureStore() for these different store parts.  
\* Now the store is ready to use.  
\* **To use it, we can simply go to the place we need it**.  
  
\* useStore() gives us access to the global store and the global dispatch function.  
\* Here I’m not interested in the dispatch function, so I can just access the state.  
\* Now if we check the page, we still see all our products there.  
\* Now let’s make the “Favorite” button work again.  
  
\* Here I’m only interested in the dispatch function.  
\* Now the “Favorite” button works again.  
\* Let’s make the “Favorites” page work next.  
  
\* This is now working.  
\* I’m totally aware that this is kind of hard to wrap your head around at first.

**Custom Hook Store Summary**  
\* We have some variables which are not global, not registered on the window object and outside of our Custom Hook but which do exist in the store.js file here and they only exist once in our application lifetime, so they’re shared in the entire application.  
\* Every other file which imports from the store.js, will use the same values.  
\* Then in the same file we create our own Custom Hook   
\* We’re not just sharing the Hook logic but also sharing the data - because they’re defined outside.  
\* Sometimes you don’t want this but here we absolutely do want this because this allows us to globally manage some state, some actions and listeners which are interested in state changes which in turn are triggered by actions.  
  
\* In our useStore() Hook we’re managing all of that. We’re having our dispatch() function in there which makes sure that whenever we call dispatch(), we update our global state and we call our listeners where **our listeners are in the end just setState() calls where we’re using that feature so that when you call that state updating function that useState() gives you, any component that uses this Hook, will re-render**.  
\* We register 1 listener per component with the help of useEffect() and we unregister it when that component is destroyed.  
  
\* The Custom Hook returns:  
  
\* And we then also have a way of initializing our store which you can call multiple times because we’re not replacing our global state or replacing our actions - instead, we’re always taking the current global state and the current actions, to merge in new data.  
\* We’re doing this so that you can create concrete store slices just as we’re doing it with Redux with multiple Reducers where in one slice you manage your products for example, maybe in another slice you manage the user authentication status. Of course you have to avoid name clashes.  
  
\* Then in configureStore() we’re setting up some actions for example.  
\* And then we call initStore() to pass our actions and our initial state for this slice of the global state so that will be merged in our global state and in our global actions.  
  
\* And then from anywhere in our project, we can use that store and then either tap into our state or use the dispatch function to dispatch an action.  
  
\* **That’s roughly kind of also the idea how Redux works behind the scenes**.  
\* Now now we rebuilt it totally without Redux, saving that extra dependency and using React and Hooks only.

**Optimizing the Custom Hook Store**  
\* There’s 1 additional thing we can do.  
\* **If we now click “Favorite”, the ProductItem renders 4 times as well, this should only render 1 time - just for the product we click**.  
=> To avoid that these other items re-render you could of course say we could wrap them in **React.memo()** - this should make sure they don’t re-render if their props didn’t change and the props for the other items certainly didn’t change, they have the same title, the same favorite status and so on.  
  
=> **If we do that, they still re-render**.  
=> The reason for that is our Custom Hook.  
=> In each ProductItem I’m using useStore() Custom Hook and in that Custom Hook, we use useState(), so whenever setState is called, the component that uses this Hook, will be re-rendered.  
=> We can improve this - the useStore() can take a `shouldListen` argument, true by default and we can use that to determine whether we want to register a listener for this component or not.  
\* Because if we have a component which only uses our store to dispatch actions like our ProductItem is going it, well then we don’t want to listen to changes inside of this ProductItem component, we’re not interested in state changes. I’m using using the Hook so that I can dispatch actions.  
\* In the useStore():  
  


\* Now we can do this:  
   
\* So now this component should not register a listener in my global listeners array and therefore it shouldn’t rebuild when our store changes because here I’m really not interested in store changes.  
=> The store change about this single product which I set to be “Favorite” is reaching this item anyways - because that is something I’m listening to in Products.js where I rebuild this list of products when I mark something as “Favorite” and therefore this specific product item will get a new prop anyways, that will go through React.memo() and therefore this single item will rebuild but the other items shouldn’t.  
\* Now we only see 1 rendering console log if I mark an item as “Favorite”.  
\* So it’s a tiny optimization which we can also put in place here and which also hopefully shows you the power of Custom Hooks and of this custom state management system - you can really optimize this for your requirements and your needs.

**Bonus: Managing Multiple State Slices with the Custom Store Hook**  
\* In this module, I show how to manage one slice (i.e. one kind of) state with the custom store hook.  
\* You can also use it to manage multiple, independent state slices though.  
\* Attached, you find an enhanced example where more than one state slice gets managed.  
\* **counter-store.js**  


\* **index.js**  
  
\* **counter.js**

**Wrap Up**  
\* **This was a really advanced module**.  
\* You saw this project with Redux, with Context API and with Custom Hook store.  
\* **What should you use?**  
\* You can definitely stick to Redux, the overhead of having an extra library is probably not that bad especially in bigger projects.  
\* And if you learned how to use it, then using it is pretty straightforward and all the heavy lifting is done for you.  
\* Now you can still switch away if you want to try something new or if you really want to save every byte you possibly can so you don’t want to add that extra Third-party dependency or maybe you don’t want that dependency to be independent from the development of that package.  
=> **Then you can use the context API for low frequency updates**.  
 => For things like authentication or themes  
=> **For high frequency updates you can use Custom Hooks**.  
\* Attached to this video, you’ll actually find an npm package which basically follows a similar idea which you could import into your project - so you again have a Thirdy-party dependency of course but one which then in turn only uses React and React Hooks to basically give you that code out of the box, so that you don’t have to write it on your own.  
**npm Package (Global State Hook)**  
<https://www.npmjs.com/package/use-global-hook>  
\* I find it very interesting to see this approach when we used a Custom Hook, I find it very interesting to see how you can manage your data with only React and JavaScript and therefore I definitely want ot share this with you.  
\* **Maybe this is the future of state management in React, it’s an early concept, an early idea, it absolutely works, you can optimize its performance as we did with `shouldListen` and therefore this is definitely something you should be aware of and you might want to try out in your project**.

**Useful Resources & Links**  
\* These resources might also be helpful:  
\* **More on React Hooks:** <https://reactjs.org/docs/hooks-intro.html>   
\* **npm Custom Store Hook package:** <https://www.npmjs.com/package/use-global-hook>

**Resources**  
npm Package (Global State Hook)  
<https://www.npmjs.com/package/use-global-hook>