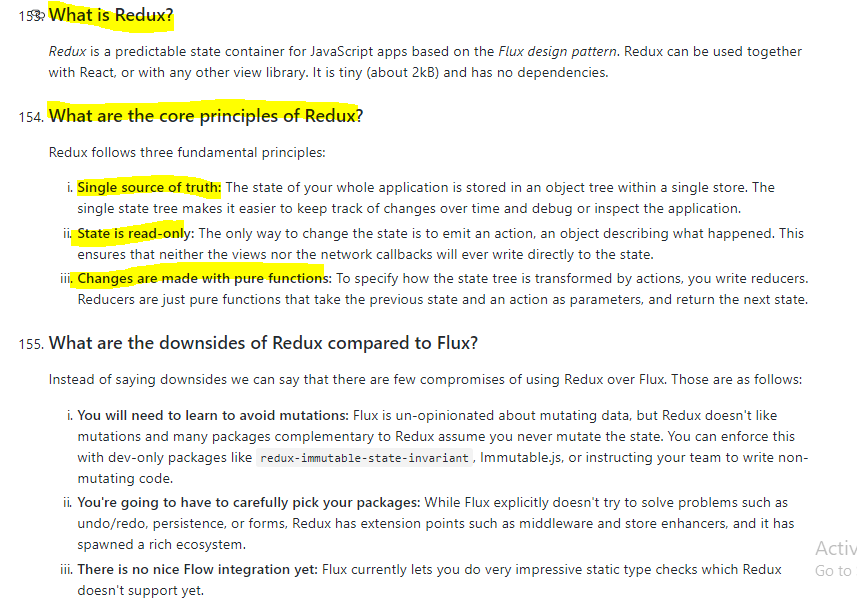
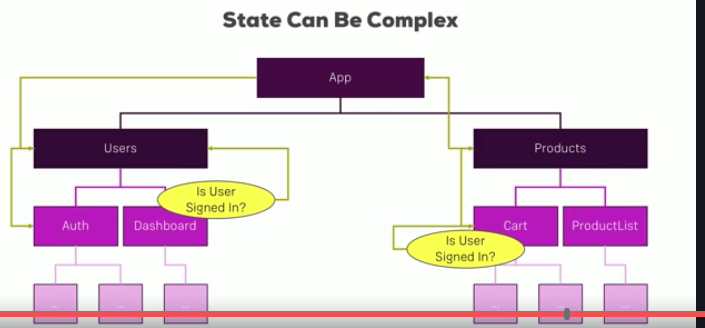
# **Redux**

it's a standalone third party library. It is a library often used in react projects though to make state management, the management of application state easier because that can be hard in more complex react projects.



**react is great at reacting to state changes and updating the UI accordingly but managing that state can get very difficult as our application grows.**

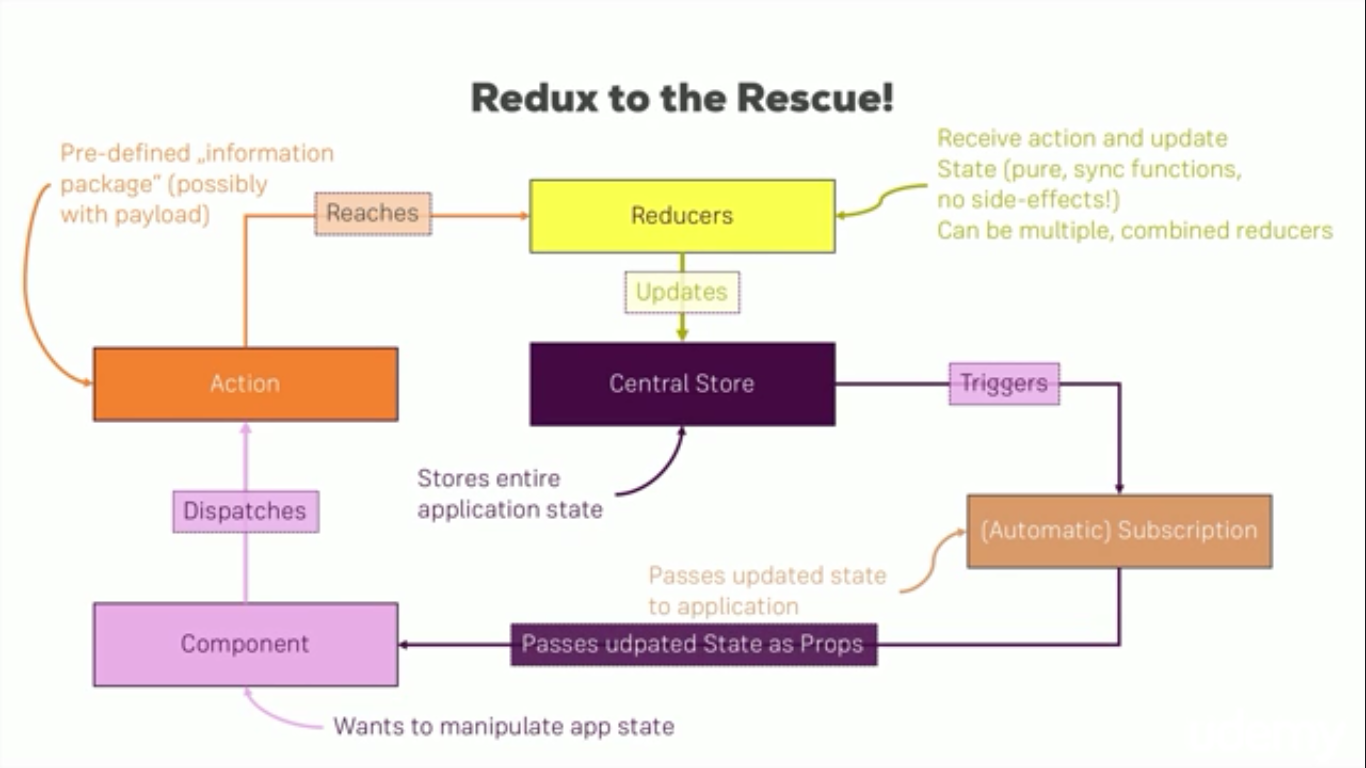


**The problem now is what if we all need that information in a totally different area of our app like in the burger builder where we need ingredients in a totally different area of the app in a checkout. If we need information there, we somehow have to create a connection between the auth component and our card component here.**

**Well that is super complex and a very long chain of props or query params or however we manage to pass data around.**

**So we need to have a clear, predictable process of updating the state on which we can rely on and which is the only process that can change our state.**

**That is actually what redux is all about, having a clearly defined process of how your state may change.**



**The first building block besides the central store are actions which are dispatched from your javascript code, in a react app, they are dispatched from within your components. And action is just information package in the end with a type, something like addIngredient or removeIngredient,so a description you could say.**

**Possibly, it also holds a payload, for example if the action is addIngredient, we need to also pass the information which ingredient and that would also be a part of the action.**

**So it's a information package we're sending out to the world or to redux to be precise, that action doesn't directly reach the store, that action doesn't hold any logic, it doesn't know how to update the store, it's just a messenger.**

**The thing changing the store is a reducer. Now here I've written reducers because we actually can combine multiple reducers into one but in the end, you'll end up with one root reducer which is directly connected to your store in the end.**

**So the action reaches the reducer and since the action contains a type, the reducer can check the type of the action, for example if it's addIngredient and we then define the code for that type of action in the reducer.**

**The reducer in the end is just a pure function which receives the action and the old state as input and which then spits out an updated state.**

**The important thing is that the reducer has to execute synchronous code only, no asynchronous code, no side effects, no HTTP requests, nothing of that, you'll learn later how you can still implement asynchronous code but in reducers, it's just input in, output out, nothing in between, no delay.**

**So this is the reducer and the reducer spits up the updated state which then is stored in the store again and replaces the old state and that has to be done in an immutable way, so we always return a new state which can be based on the old one but which is technically a new javascript object, because objects are reference types in Javascript and we want to make sure that we don't accidentally change the old one.**

**So that is how the reducer handles the action,**

**now the store is up to date.**

How do we get the updated state back into our component then? For that, we use a subscription model.

**The store triggers all subscriptions whenever the state changes, whenever the state is updated in the store. And of course our component can subscribe to store updates and it then receives that update automatically, this is how simple it is.**

**It works through a subscription model and we simply say hey I want to get notified whenever the state changes, just as we say hey I want to change the state, here is an action describing my plans.**

## **Subscribe : Connecting Store to react**

**I need a special package because redux alone is standalone, it is not connected to react.**

**I'll install it with npm install --save react-redux**

**it allows us to hook up our redux store to our react application.**

let's **import something from react-redux, its call connect**



**it's a function, a higher order component actually, a higher order component we use on the export.**

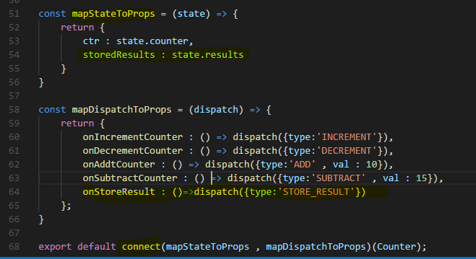
**Actually connect returns a function which takes then a component as input, so connect is not really a higher order component, it's a function which returns a higher order component.**

Precisely, **we passed two pieces of information to connect.**

**(1) Which part of the whole application state is interesting to us because here we only have counter but in bigger apps, you may have loads and loads of different states and pieces of states you manage and you don't need all of that and all your containers. So you can define which slice of the state do I want to get in this container.**

**(2) which actions do I want to dispatch because again, in bigger applications you may have thousands of actions dispatched from all over the application but a given individual container may only dispatch a couple of these.**

**So the actions we want to dispatch and the state we want to get**

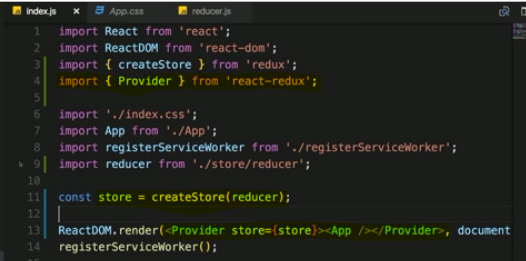


## **Provider**

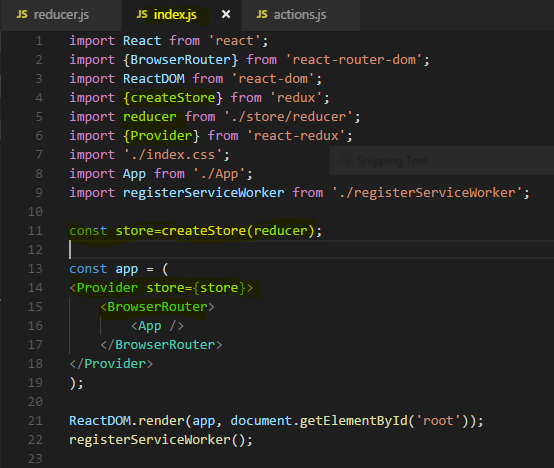
**we wrap our app component with Provider. Provider is a helper component which allows us to kind of inject our store into the react components.**

**For hooking up the provider component with our store here, I need to set up a property, a special property expected by the Provider component. It's named store**

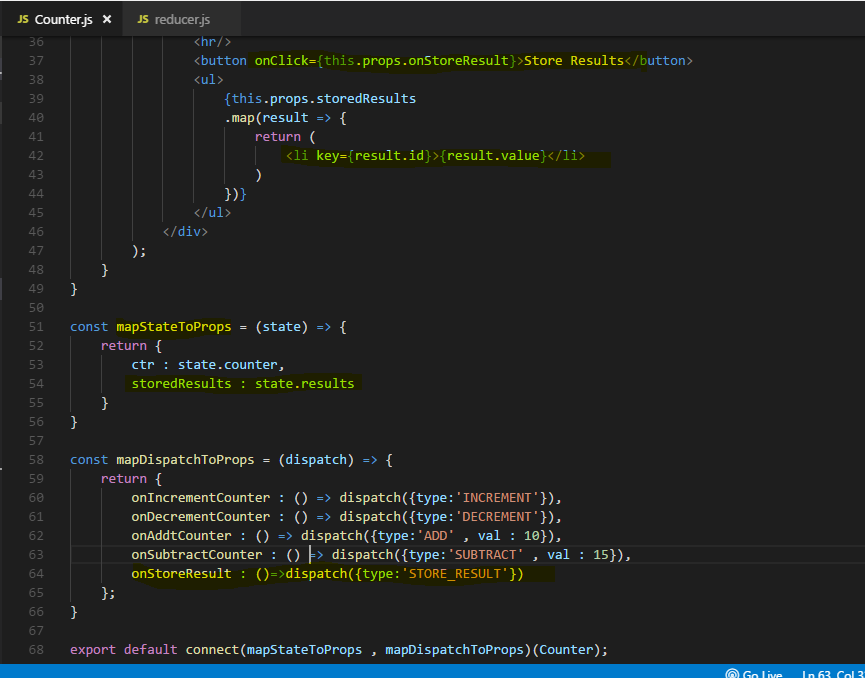
**So I'm passing that store constant as a value to this prop.**



OR



**So whenever you click on increament button this action will be dispatched to Store via connect and we get updated state as props because of subscription of this Action and that eventually re render the react application**

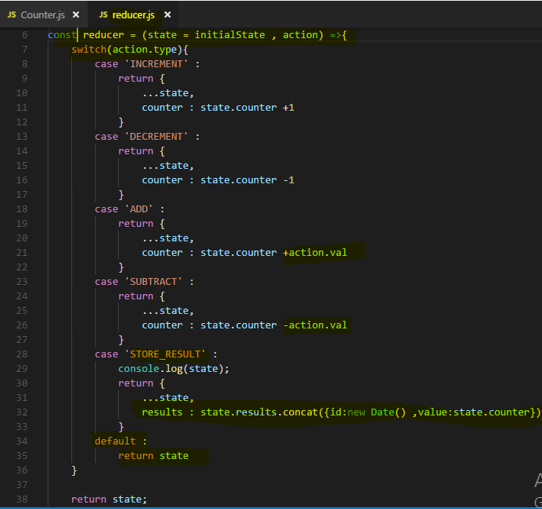
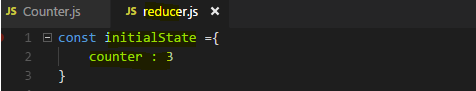


In reducers

**one important thing, you don't set state.counter++ which would increment it and return state because this is not immutable, you're mutating the original state here.**

**So what you instead do is you return a new javascript object where you may first copy the old state with the spread operator, state and then overwrite the one property you want to adjust, so the counter and if that also would be a javascript object, you would have to copy it first too so that you never mutate any data, never, always do this immutably.**

Note : **the new object we return from Action in Reducer, it updates the state of Reducer behind the scene**



**Therefore if that reducer doesn't handle that action type you dispatched, you have to return the current state to not break your application.**

**Important Note :**

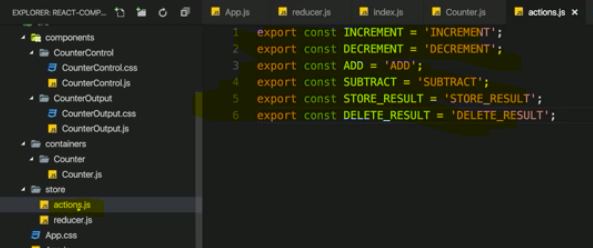
**Whenever we click on Store result button , STORE\_RESULT action triggered and it returns a Updated state to the Component in subscription mapStateToProps as storedResult available as a props to this component.**

**Then we use this data to display our <li> element**

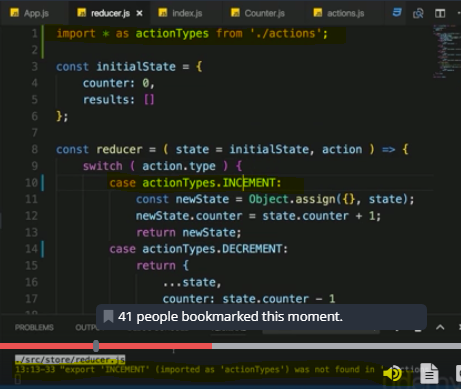
**Important Note :**

**The new object which we return from an Action in Reducer , updates the state of Reducer behind the scene. So Reducer state gets updated by the action**

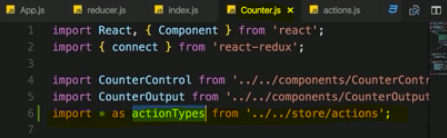
## Outsourcing Action types

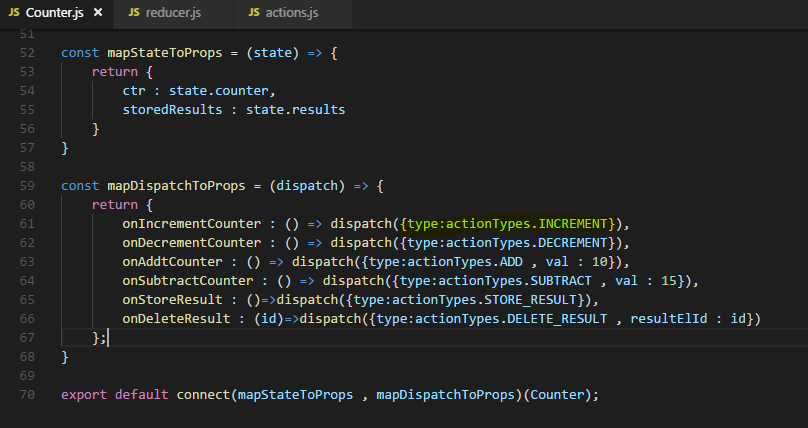


we can now simply import everything from a given file and store it in some javascript object



The huge advantage of this approach simply is we're now importing our constants here and if we mistype one constant name, we'll actually get an error by our IDE or by that build process then I do the same where I dispatch them in component, it of course doesn't make any sense to still have hardcoded values here because I could still mistype,



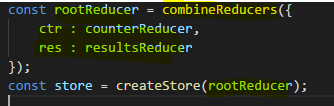


**we're now always referring to that property, to the constants we are exporting in the actions file and hence we can't mess up by mistyping or having different identifiers because now we have only one place where we set up and store our identifiers**

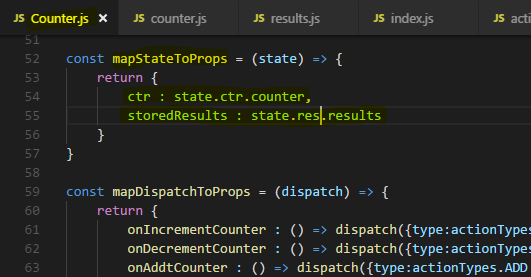
## **combineReducers**

**We can create multiple reducer which their local state for counter and results. At the end it all combine to one root reducer hence one global state.**

**Now to combine them, I need to import a helper function from the redux package, it's called combineReducers, as the name suggests, this is a function which takes a javascript object mapping our reducers to different slices of our state as input and merges everything into one state and one reducer for us.**



**So in the counter.js file, if we want to access the counter we have to access the global state.ctr.counter. since this is the name we gave this slice of our global state and for the result, it would be state.res.results**



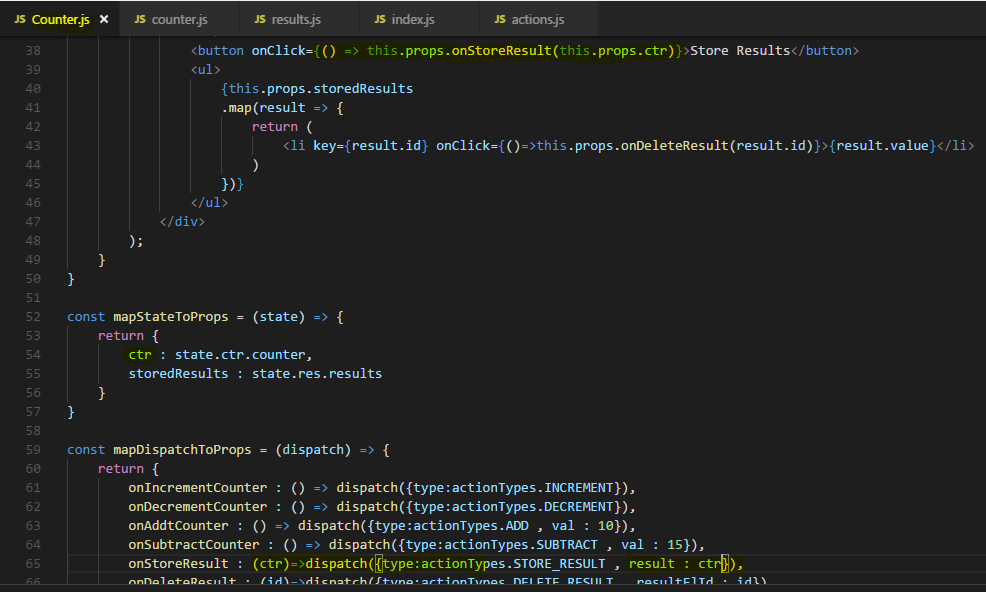
**Note : inside a reducer function, it basically has no access to the global state only to that state of that reducer function.**

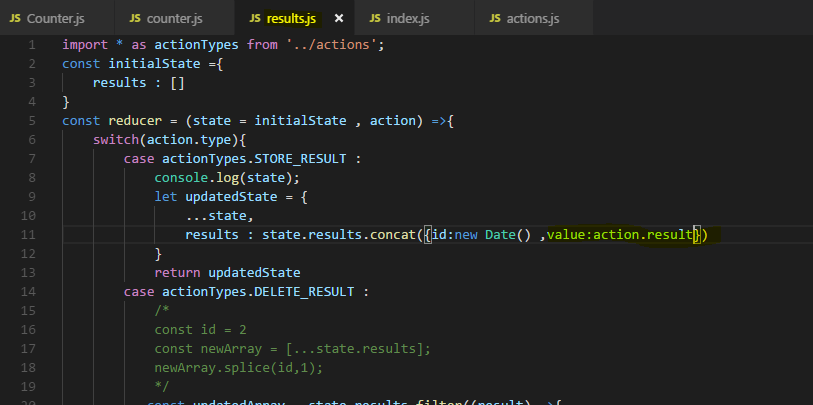
**That's different than the counter component where we connect our react component to the global state, there we can access the different pieces of the state through our slices we set up in index.js**

**This does not work inside of the reducers.**

**So if we are in a reducer where we need to get a value from the global state, we should simply get it as an action payload and this is generally how your reducers work anyways**

**most of the time, it's old state plus action plus optionally action data and you return a new state.**





## **Combine Local UI State and Redux**

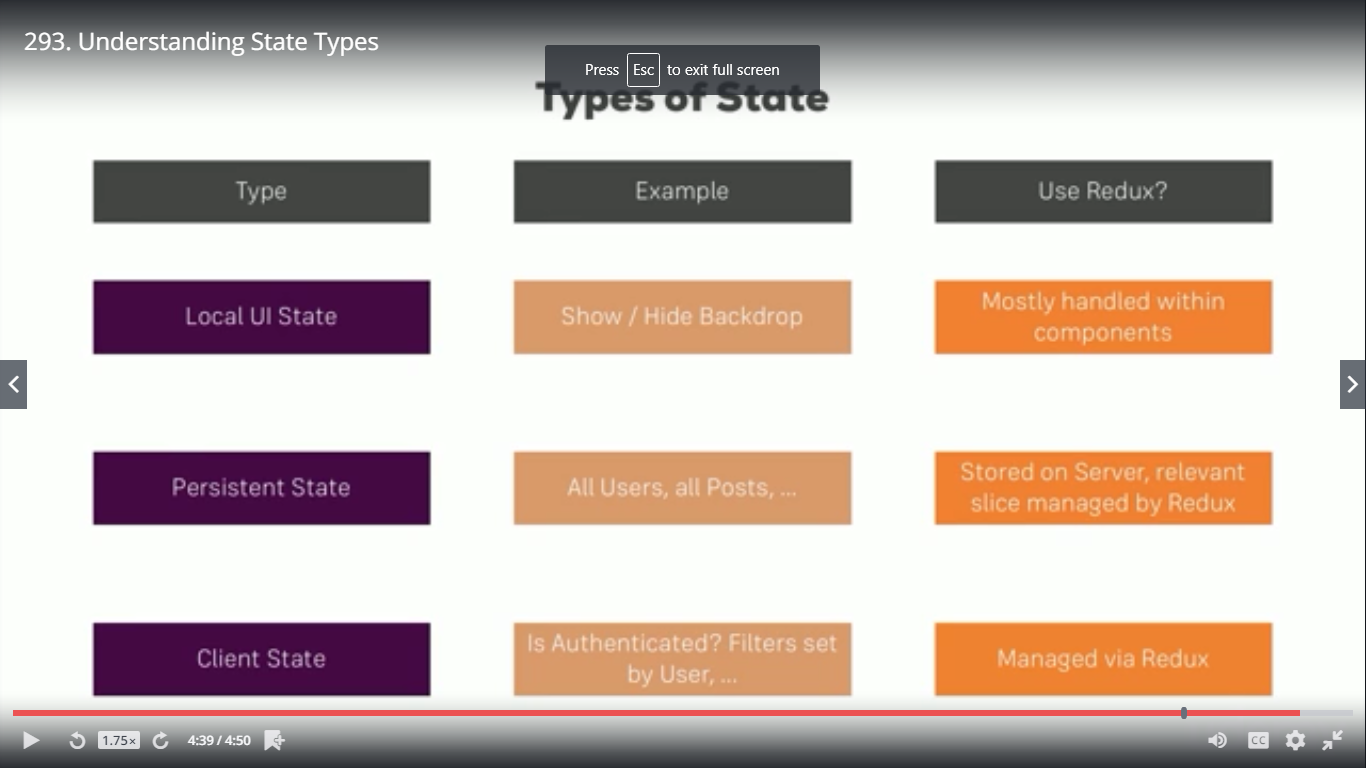
**Some data of component are local UI state, Whatever the user entered into these inputs here probably isn't relevant to the entire application, there is no need to store this in the global Redux store, you can absolutely store it in the state of that component because it only matters to that component, it does matter to the other components as soon as this button is clicked.**

**Hence combine both Local UI state and redux for your application.**

**They can co-exist together with redux, you can put everything into redux but you don't have to.**

**For any decent medium size or big application, using redux and managing the state there is probably a good idea but then still, we have to ask which state should be used for redux because you shouldn't necessarily manage all the state in it.**

Let's have a look at the various types of state, some examples and whether you should use redux for them or not.

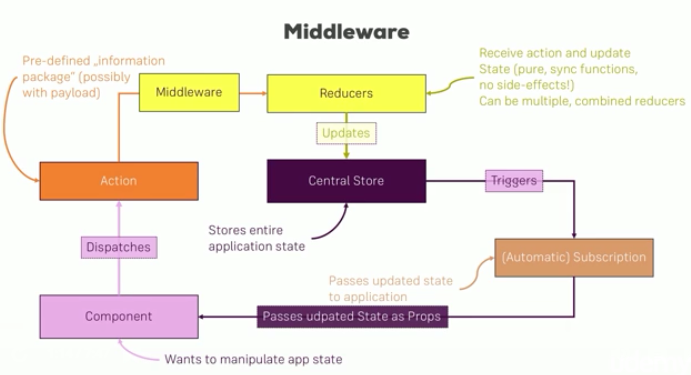


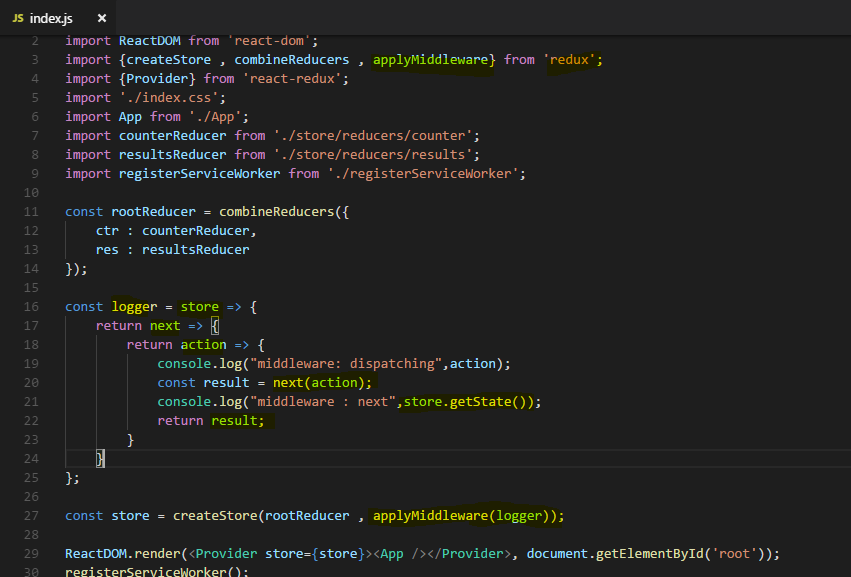
**redux of course is just for managing the state in your application as long as your application is alive. And always keep in mind, when the user refreshes your page, your state is gone. so redux is not a replacement for a database,**

## **Adding Middleware**

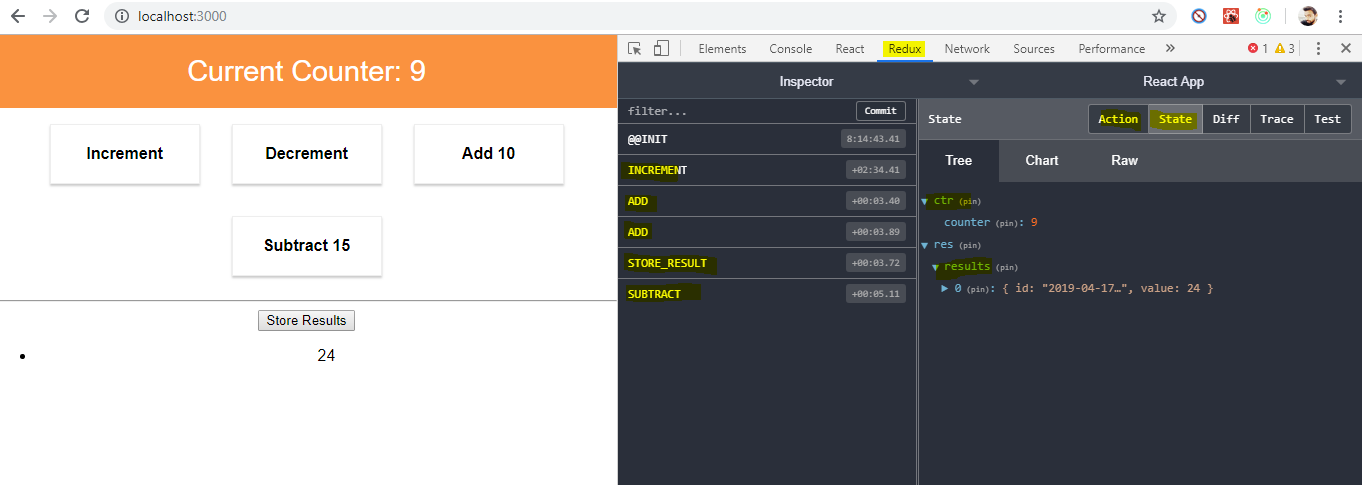
**You may add middleware to it right between your action being dispatched and it reaching the reducer, this is where you can add middleware.**

**So we can add middleware and the action will still reach the reducer thereafter but we can do something with that action before it reaches the reducer, that can be simply logging something**





## **Redux Dev Tools**



**Here we can see , we get details of actions we are performing on application and the state changes dynamically so we are getting updated state .**

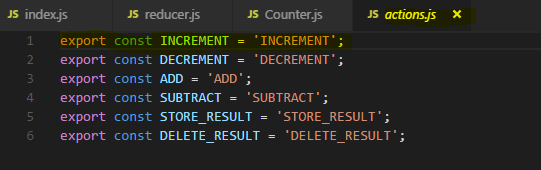
**we can always see how we adjusted our state and what our current state**

**is and what our state was in the past**. So redux dev tools are extremely useful

## **Action creators : Executing Asynchronous Code**

**We can execute asynchronous code with the help of so-called action creators,**

**An action creator is just a function which returns an action or which creates an action**



//export const INCREMENT = 'INCREMENT';

export const increment = () => {

return {

type : 'INCREMENT'

}

}

//export const DECREMENT = 'DECREMENT';

export const decrement = () => {

return {

type : 'DECREMENT'

}

}

//export const ADD = 'ADD';

export const add = (payloadData) => {

return {

type : 'ADD',

...payloadData

}

}

//export const SUBTRACT = 'SUBTRACT';

export const subtract = (payloadData) => {

return {

type : 'SUBTRACT',

...payloadData

}

}

//export const STORE\_RESULT = 'STORE\_RESULT';

export const storeResult = (payloadData) => {

return {

type : 'STORE\_RESULT',

...payloadData

}

}

//export const DELETE\_RESULT = 'DELETE\_RESULT';

export const deleteResult = (payloadData) => {

return {

type : 'DELETE\_RESULT',

...payloadData

}

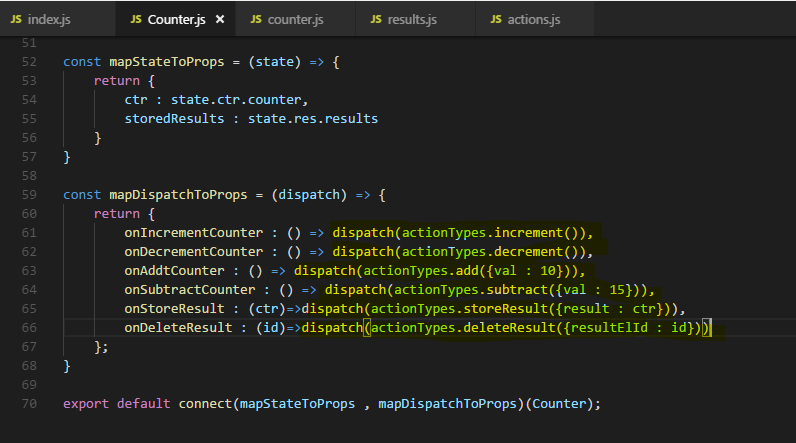
}

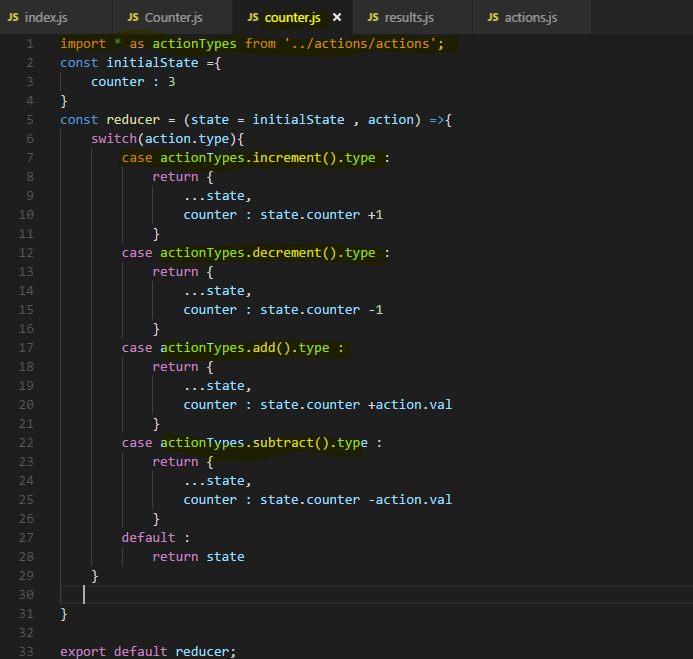
**Now we can use these function for different actions.**

**it will return something eventually and it will receive something, it can receive something I should say, it receives any payloads you want to pass with that action.**

**For other action creators which receive any payload we should pass them and receive inside the action creator function and return this also.**

import \* as actionTypes from '../../store/actions/actions';





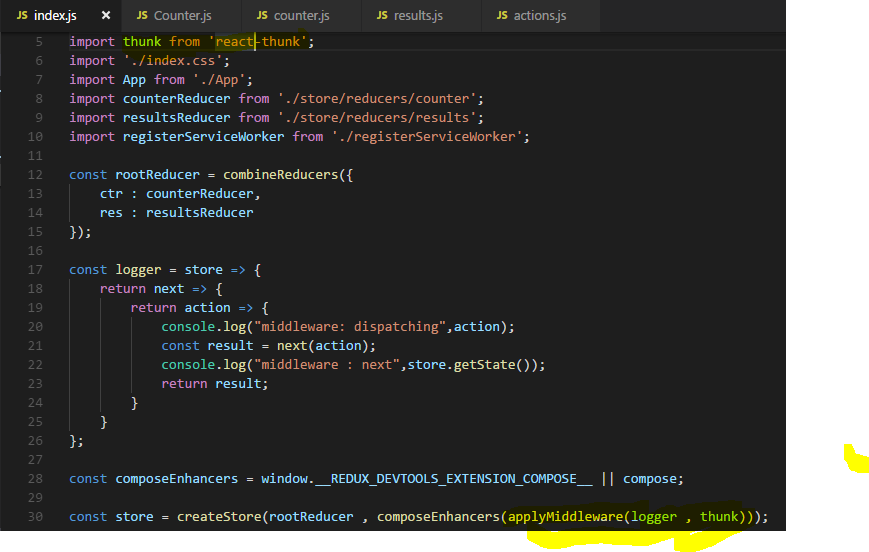
## **Redux-thunk - Handling Asynchronous Code**

I want to take advantage of action creator to handle asynchronous code and 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**.

**React-thunk is a library which as I just said adds a middleware to your project which allows your actions to not or your action creators to be precise 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,**



OR



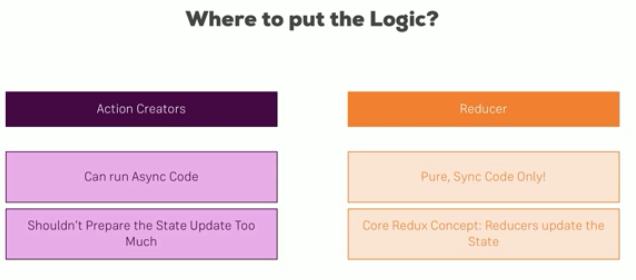
**the function receives dispatch as an argument, the dispatch action, now we get dispatch in here due to redux-thunk. I said that middleware runs between the dispatching of an action and the point of time the action reaches the reducer, now the thing we do here is we still dispatch an action but then redux-thunk, the middleware comes in, steps in, has access to the action there, basically blocks the old action we could say and dispatches it again in the future.**

**Now the new action will reach the reducer but in-between, redux-thunk is able to wait because it can dispatchan action whenever it wants.**

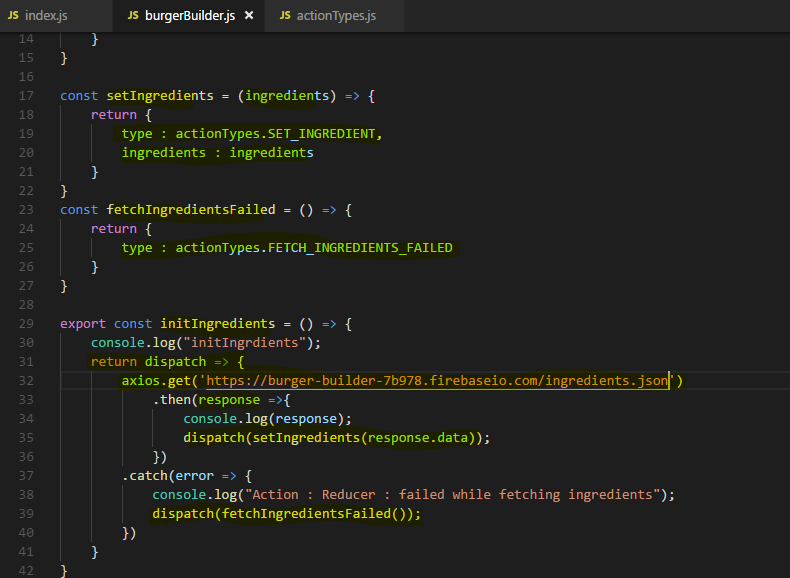
**This is the asynchronous part and that is exactly allowing us to execute some asynchronous code inside of this function.**

**inside of this function passed to set timeout, we can execute dispatch to now dispatch whichever action we want to dispatch.**

**So what do we typically do is we create asynchronous action creators, which in the end dispatch actions created by synchronous ones.**



**action creators as you learned are great for running async code when you dispatch an action, reducers on the other hand only are able to run synchronous code and are pure, input in updated state out.**

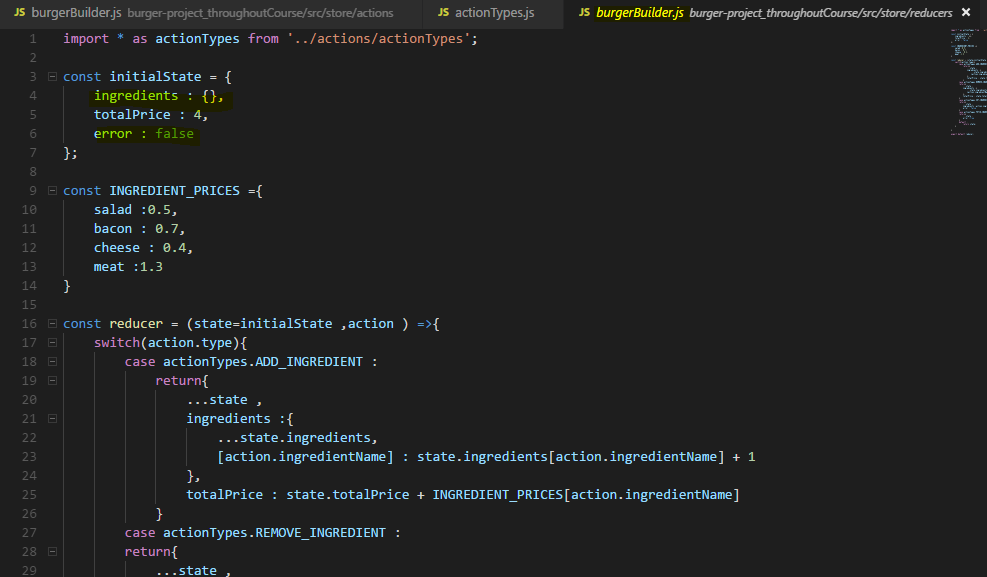


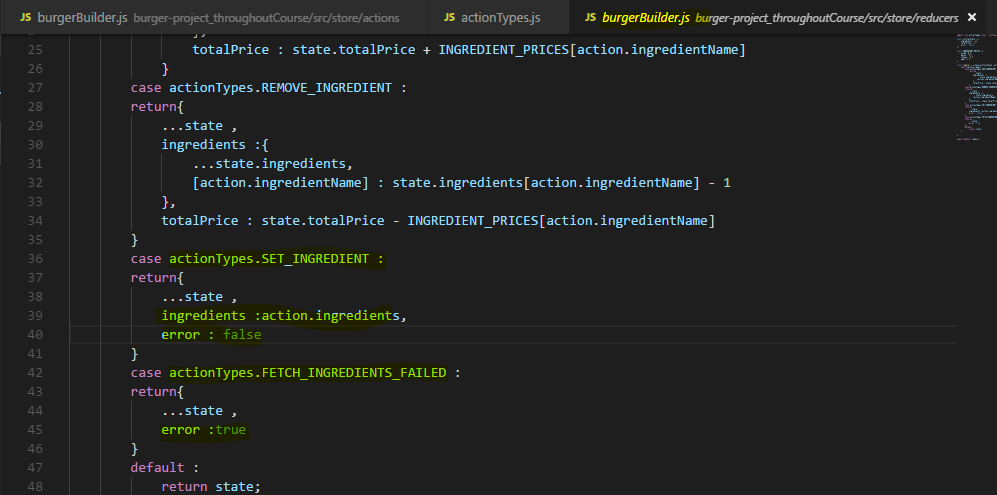
**Here we define an action creator initIngredients , there we do asynchronous call to get ingredients from server. We can do this because of middleware redux-thunk , which blocks the dispatch until asynchronous call doesn’t complete and then dispatches an action which finally reach to Reducer.**

**If ajax call succed then we dispatch an action setIngredients with type :“SET\_INGREDIENT” and it will finally reach to Reducer and update the state.**

**If ajax call fails then we dispatch another action fetchIngrediensFailed with type:”FETCH\_INGREDIENTS\_FAILED” and it will reach to reducer to update the state.**

**Lets see the updated Reducer**

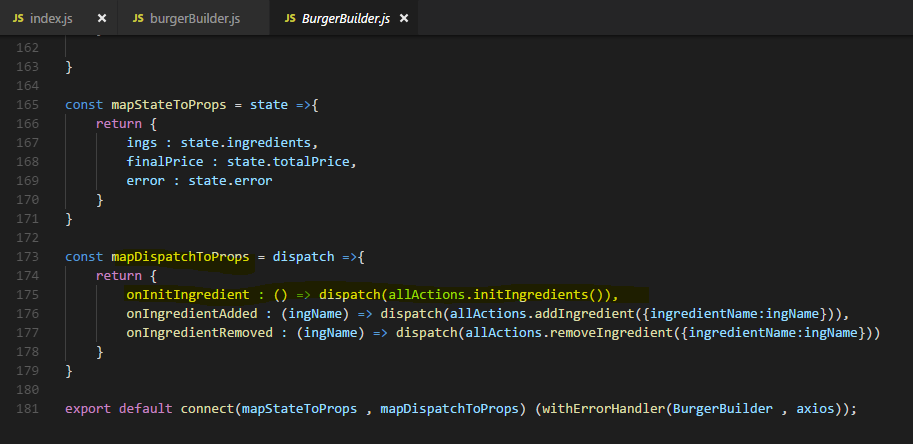




**So I also want to be able to dispatch that from the burger builder**

**So I'll add a new property here to mapDispatchToProps to Dispatch initIngredients Action.**

**So that it will be available as a props to this component and we can call or trigger it.**



**Now the last step is to call onInitIngredient in our componentDidMount method here,**

**Previously we call an action on button click but you know we call this anywhere.**

so there I will now execute this.props.onInitIngredient, like this and this should execute this, dispatch this action and hopefully successfully fetch our ingredients.



## **Other Interview Questions**

