JAVA FULL STACK DEVELOPMENT PROGRAM

React Session: Day 2

What is Redux?

- Redux is a predictable state container designed to help you write JavaScript apps that behave consistently across client, server, and native environments and are easy to test.
- While it’s mostly used as a state management tool with React, you can use it with any other JavaScript framework or library.
- With Redux, the state of your application is kept in a store, and each component can access any state that it needs from this store.

Why We Need Redux?

- State management is essentially a way to facilitate communication and sharing of data across components. It creates a tangible data structure to represent the state of your app that you can read from and write to.
- In an app where data is shared among components, it might be confusing to actually know where a state should live. Ideally, the data in a component should live in just one component, so sharing data among sibling components becomes difficult.
- in React, to share data among siblings, a state has to live in the parent component. A method for updating this state is provided by the parent component and passed as props to these sibling components.

Why We Need Redux?

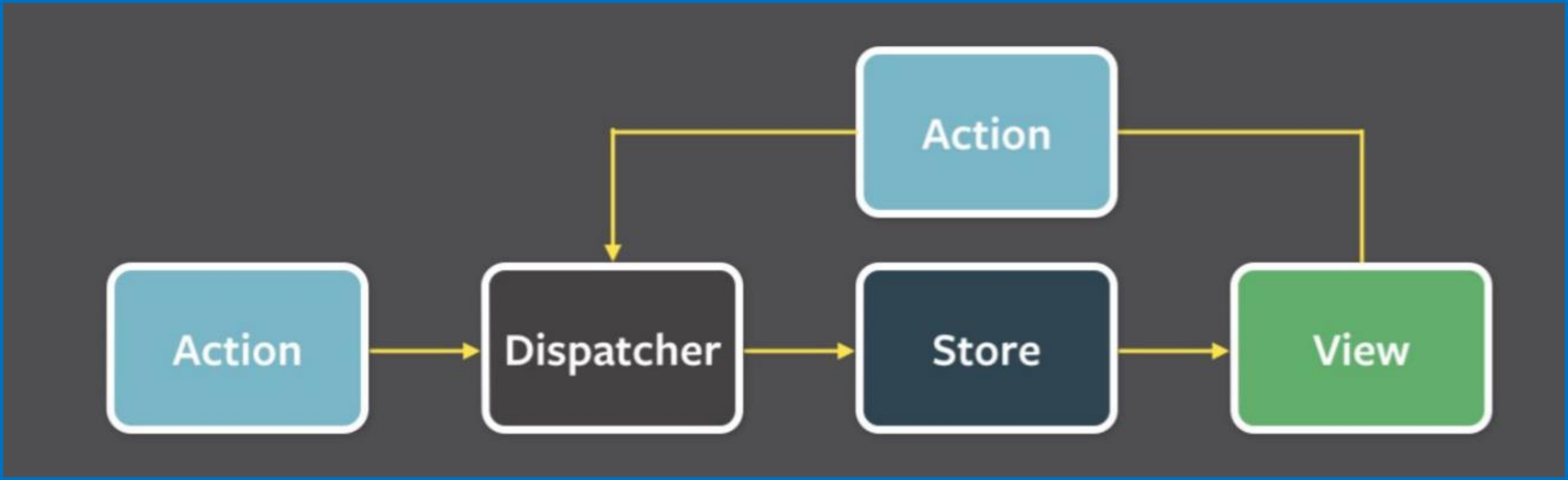
- Now imagine what happens when a state has to be shared between components that are far apart in the component tree. The state has to be passed from one component to another until it gets to where it is needed.
- It's clear that state management gets messy as the app gets complex. This is why you need a state management tool like Redux that makes it easier to maintain these states.

How Redux Works?

- The way Redux works is simple. There is a central store that holds the entire state of the application. Each component can access the stored state without having to send down props from one component to another.
- There are three building parts: actions, store, and reducers. Let's briefly discuss what each of them does. This is important because they help you understand the benefits of Redux and how it's to be used.

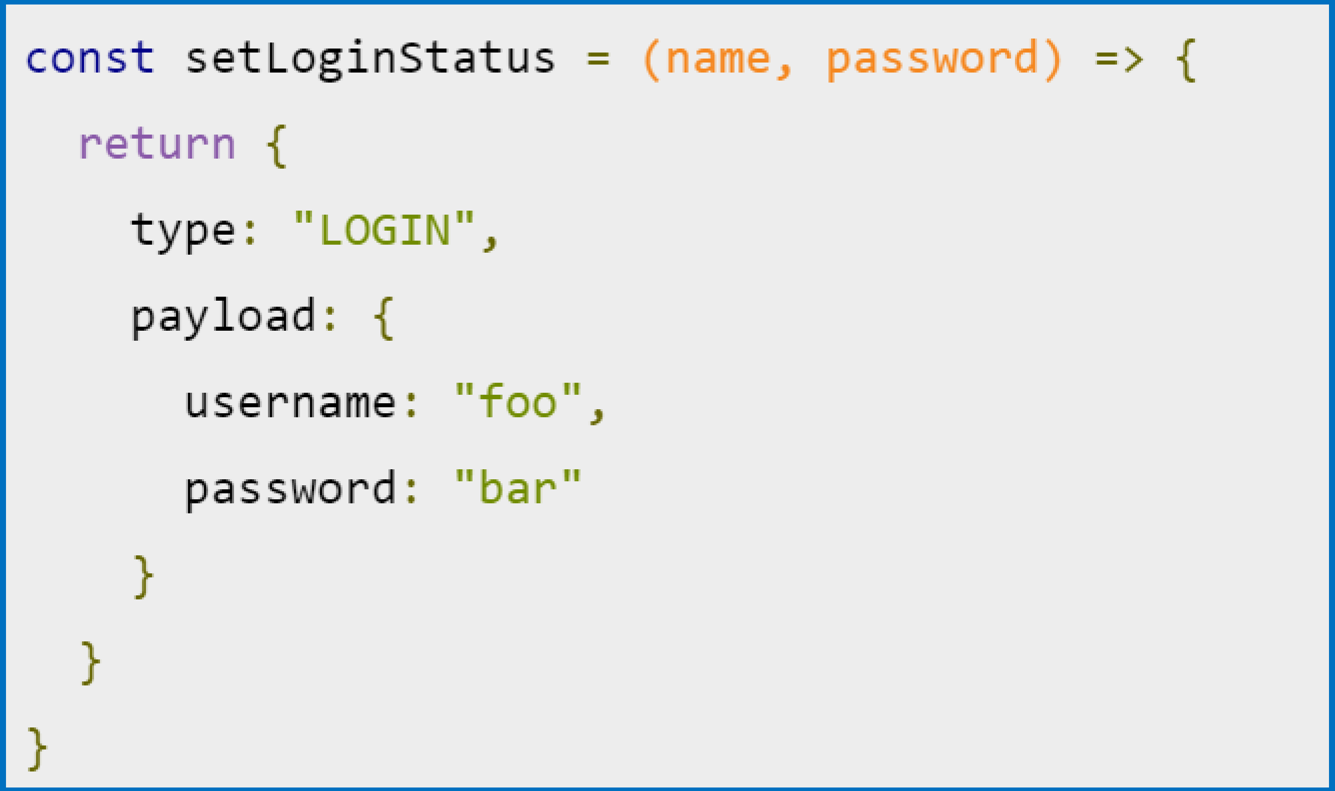
How Redux Works?

- Everything stores in the centralized store



Actions

- Simply put, actions are events. They are the only way you can send data from your application to your Redux store. The data can be from user interactions, API calls, or even form submissions.
- Actions are sent using the ***store.dispatch()*** method. Actions are plain JavaScript objects, and they must have a type property to indicate the type of action to be carried out. They must also have a payload that contains the information that should be worked on by the action.

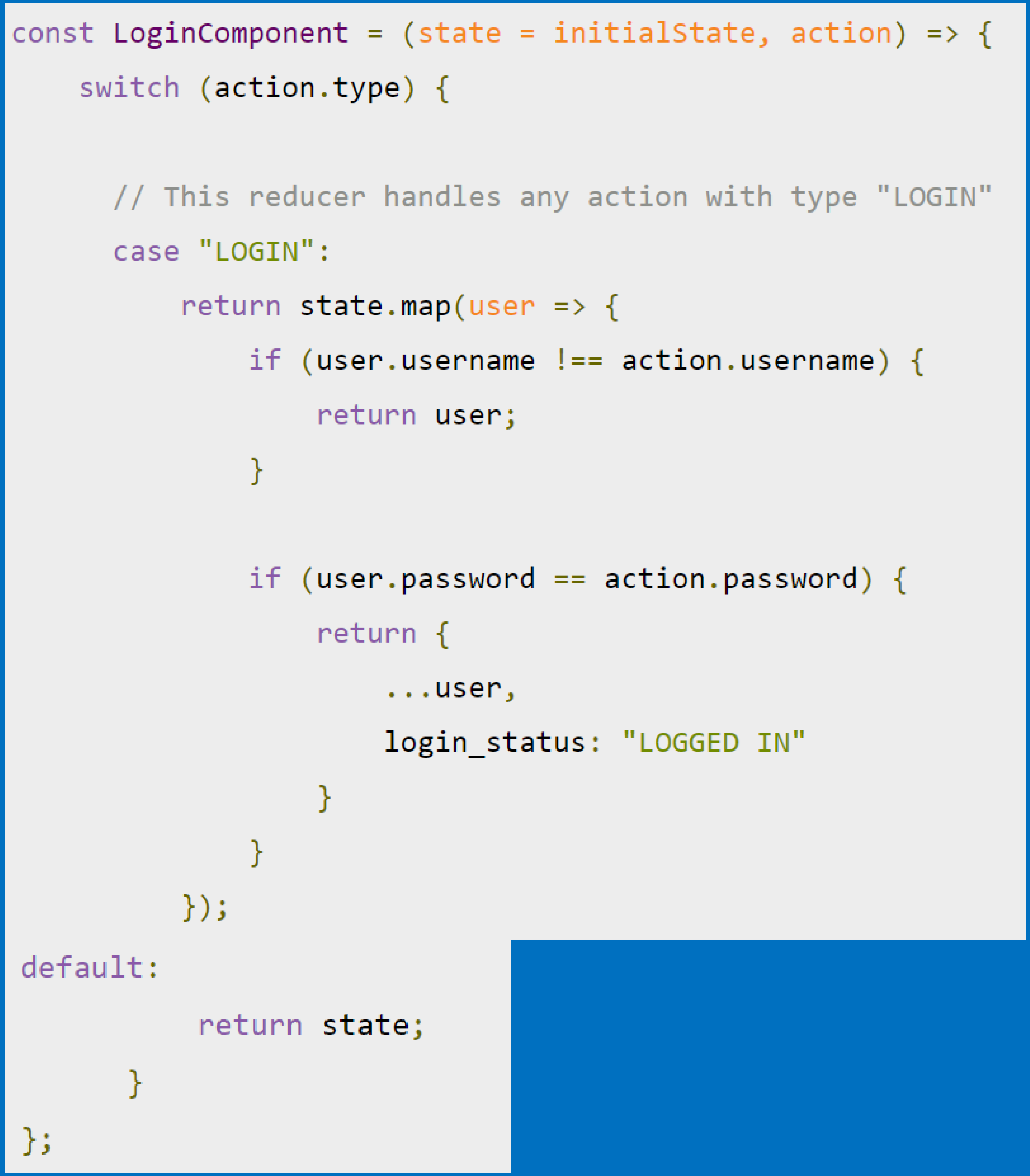


```
const setLoginStatus = (name, password) => {
  return {
    type: "LOGIN",
    payload: {
      username: "foo",
      password: "bar"
    }
  }
}
```

Reducer

- The store uses the reducer to handle actions, which are dispatched or 'sent' to the store with its dispatch-method.(they specify how the state of an application changes in response to an action sent to the store.)
- The impact of the action to the state of the application is defined using a reducer.
- In practice, a reducer is a pure function which is given the current state and an action as parameters, perform an action and returns a new state.
- In other words, (state, action) => newState.

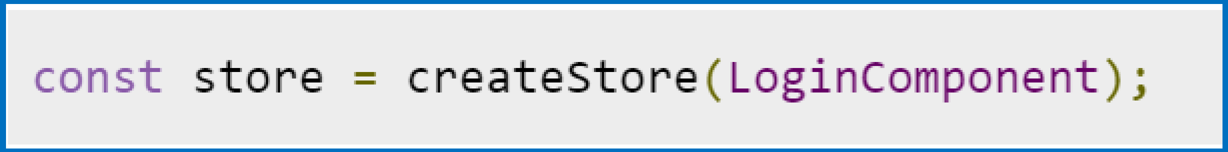
Reducer



```
const LoginComponent = (state = initialState, action) => {
    switch (action.type) {

      // This reducer handles any action with type "LOGIN"
      case "LOGIN":
          return state.map(user => {
              if (user.username !== action.username) {
                  return user;
              }

              if (user.password == action.password) {
                  return {
                      ...user,
                      login_status: "LOGGED IN"
                  }
              }
          });
default:
           return state;
       }
};
```

Store

- The store holds the application state. It is highly recommended to keep only one store in any Redux application. You can access the state stored, update the state, and register or unregister listeners via helper methods.



```
const store = createStore(LoginComponent);
```

Note: You should import createStore from ‘redux’ library

React Redux

- Installation
  - Using Create React App

```
npx create-react-app my-app --template redux
```

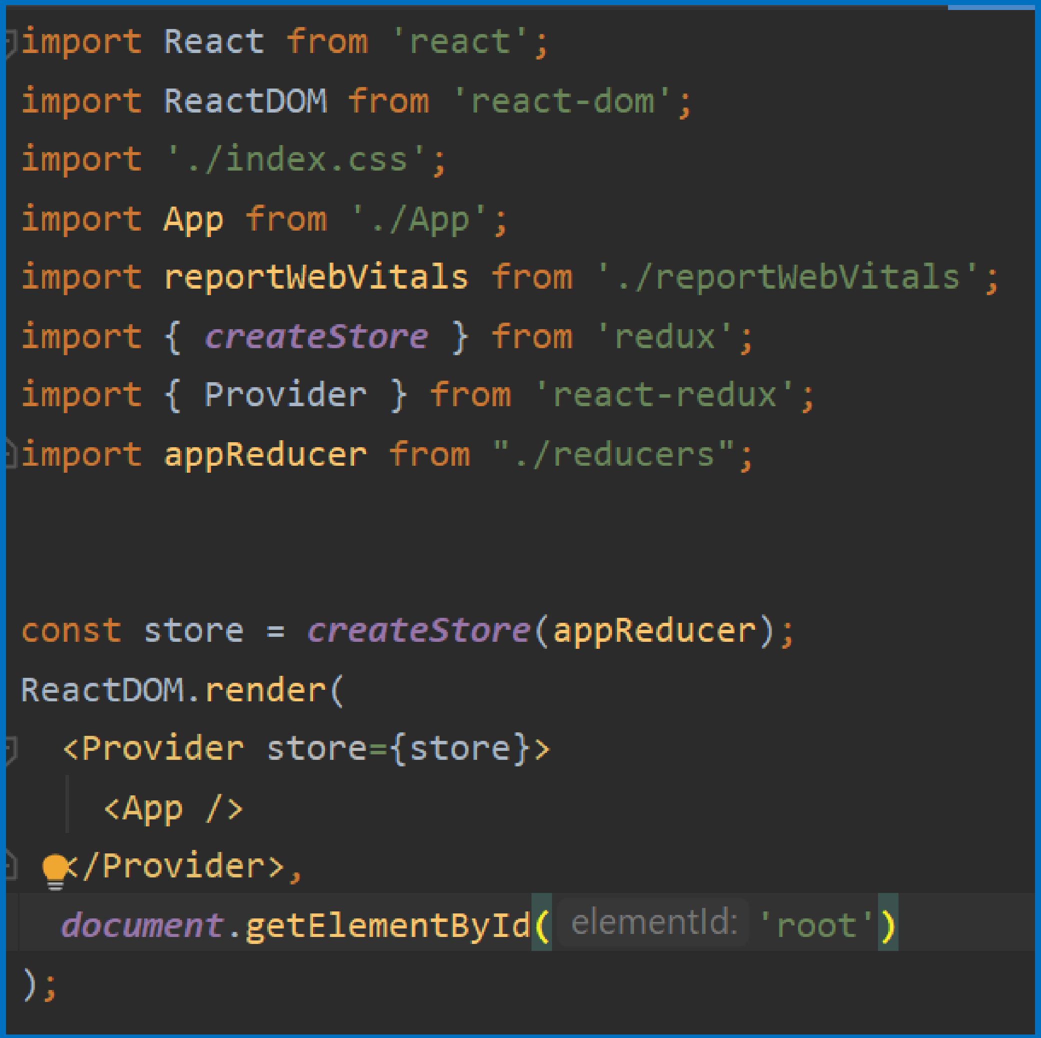
  - An Existing React App (which I prefer)
    - To use React Redux with your React app, install it as a dependency

```
# If you use npm:
npm install react-redux

# Or if you use Yarn:
yarn add react-redux
```

    - You'll also need to install Redux and set up a Redux store in your app.

Provider

- The <Provider> component makes the Redux store available to any nested components that need to access the Redux store.
- Since any React component in a React Redux app can be connected to the store, most applications will render a <Provider> at the top level, with the entire app's component tree inside of it.



```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import appReducer from "./reducers";


const store = createStore(appReducer);
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById( elementId: 'root')
);
```

Connecting the Components

- React Redux provides a connect function for you to read values from the Redux store (and re-read the values when the store updates).
- The connect function takes two arguments, both optional:
  - ***mapStateToProps:***
    - called every time the store state changes. It receives the entire store state, and should return an object of data this component needs.
  - ***mapDispatchToProps***: this parameter can either be a function, or an object.
    - If it’s a function, it will be called once on component creation. It will receive dispatch as an argument, and should return an object full of functions that use dispatch to dispatch actions.
    - If it’s an object full of action creators, each action creator will be turned into a prop function that automatically dispatches its action when called. Note: We recommend using this “object shorthand” form.

Connecting the Components

```
const mapStateToProps = (state, ownProps) => ({
  // ... computed data from state and optionally ownProps
})

const mapDispatchToProps = {
  // ... normally is an object full of action creators
}

// `connect` returns a new function that accepts the component to wrap:
const connectToStore = connect(mapStateToProps, mapDispatchToProps)
// and that function returns the connected, wrapper component:
const ConnectedComponent = connectToStore(Component)

// We normally do both in one step, like this:
connect(mapStateToProps, mapDispatchToProps)(Component)
```

mapStateToProps

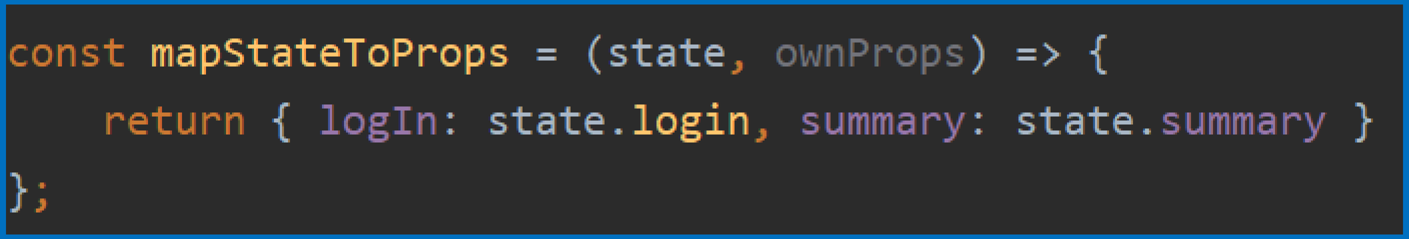
As the first argument passed in to connect, mapStateToProps is used for selecting the part of the data from the store that the connected component needs. It's frequently referred to as just mapState for short.

- It is called every time the store state changes.
- It receives the entire store state, and should return an object of data this component needs.

```
function mapStateToProps(state, ownProps?)
```

```
const mapStateToProps = (state, ownProps) => {
    return { logIn: state.login, summary: state.summary }
};
```

mapStateToProps



```
const mapStateToProps = (state, ownProps) => {
    return { logIn: state.login, summary: state.summary }
};
```

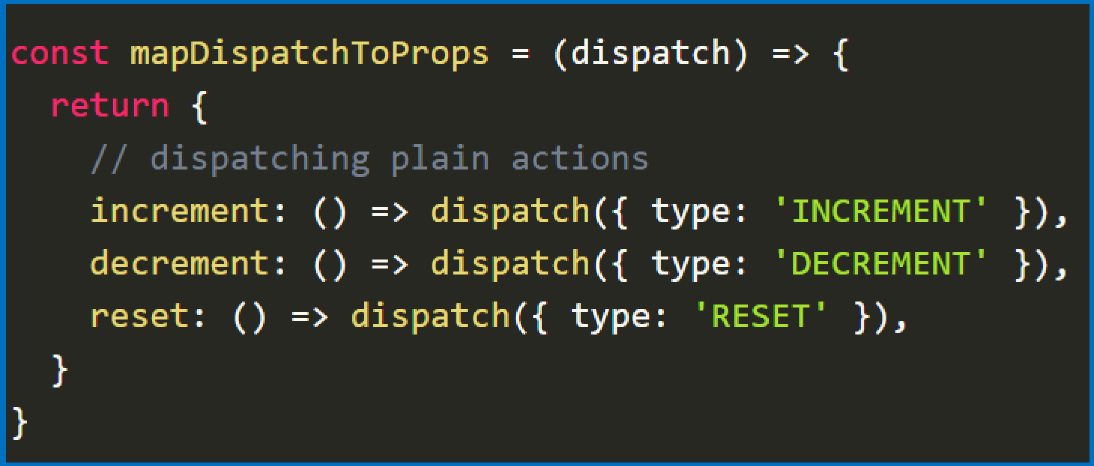
- It should take a first argument called state, optionally a second argument called ownProps, and return a plain object containing the data that the connected component needs.
- This function should be passed as the first argument to connect, and will be called every time when the Redux store state changes. If you do not wish to subscribe to the store, pass null or undefined to connect in place of mapStateToProps.

mapDispatchToProps

- As the second argument passed in to connect, mapDispatchToProps is used for dispatching actions to the store.
    - dispatch is a function of the Redux store. You call store.dispatch to dispatch an action. This is the only way to trigger a state change.
    - With React Redux, your components never access the store directly - connect does it for you. React Redux gives you two ways to let components dispatch actions:
        - By default, a connected component receives props.dispatch and can dispatch actions itself.
        - connect can accept an argument called mapDispatchToProps, which lets you create functions that dispatch when called, and pass those functions as props to your component.
    - The mapDispatchToProps functions are normally referred to as mapDispatch for short, but the actual variable name used can be whatever you want.

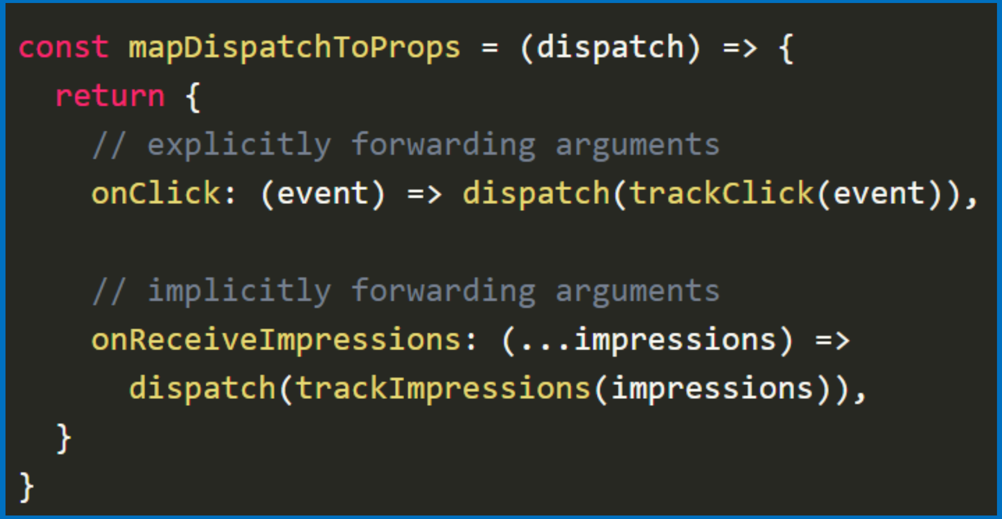
mapDispatchToProps

- Arguments: 1. dispatch 2. ownProps (optional)
- The mapDispatchToProps function will be called with dispatch as the first argument. You will normally make use of this by returning new functions that call dispatch() inside themselves, and either pass in a plain action object directly or pass in the result of an action creator.



```
const mapDispatchToProps = (dispatch) => {
  return {
    // dispatching plain actions
    increment: () => dispatch({ type: 'INCREMENT' }),
    decrement: () => dispatch({ type: 'DECREMENT' }),
    reset: () => dispatch({ type: 'RESET' }),
  }
}
```

You will also likely want to forward arguments to your action creators:



```
const mapDispatchToProps = (dispatch) => {
  return {
    // explicitly forwarding arguments
    onClick: (event) => dispatch(trackClick(event)),

    // implicitly forwarding arguments
    onReceiveImpressions: (...impressions) =>
      dispatch(trackImpressions(impressions)),
  }
}
```

mapDispatchToProps

- ownProps (optional)
    - If your mapDispatchToProps function is declared as taking two parameters, it will be called with dispatch as the first parameter and the props passed to the connected component as the second parameter, and will be re-invoked whenever the connected component receives new props.
    - This means, instead of re-binding new props to action dispatchers upon component re-rendering, you may do so when your component's props change.

React Redux Example

- How to integrate the above terms to form a complete application ?

Router

- In old school, we change the page by making a GET request and render the
- HTML representing the view returned by backend.
- In single page apps, we are, in reality, always on the same page. The
- Javascript code run by the browser creates an illusion of different "pages".
- This is achieved by using React Router

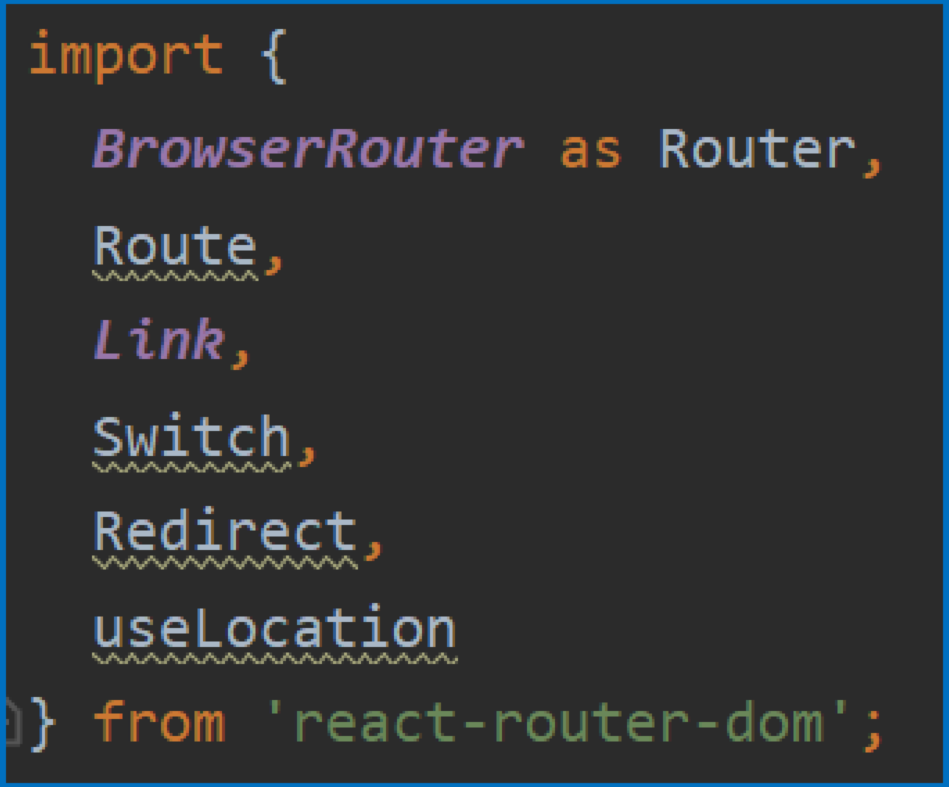
react-router-dom

- Since we're building a web app, we'll use react-router-dom in my sample
- Installation:

```
npm install react-router-dom
```

Router

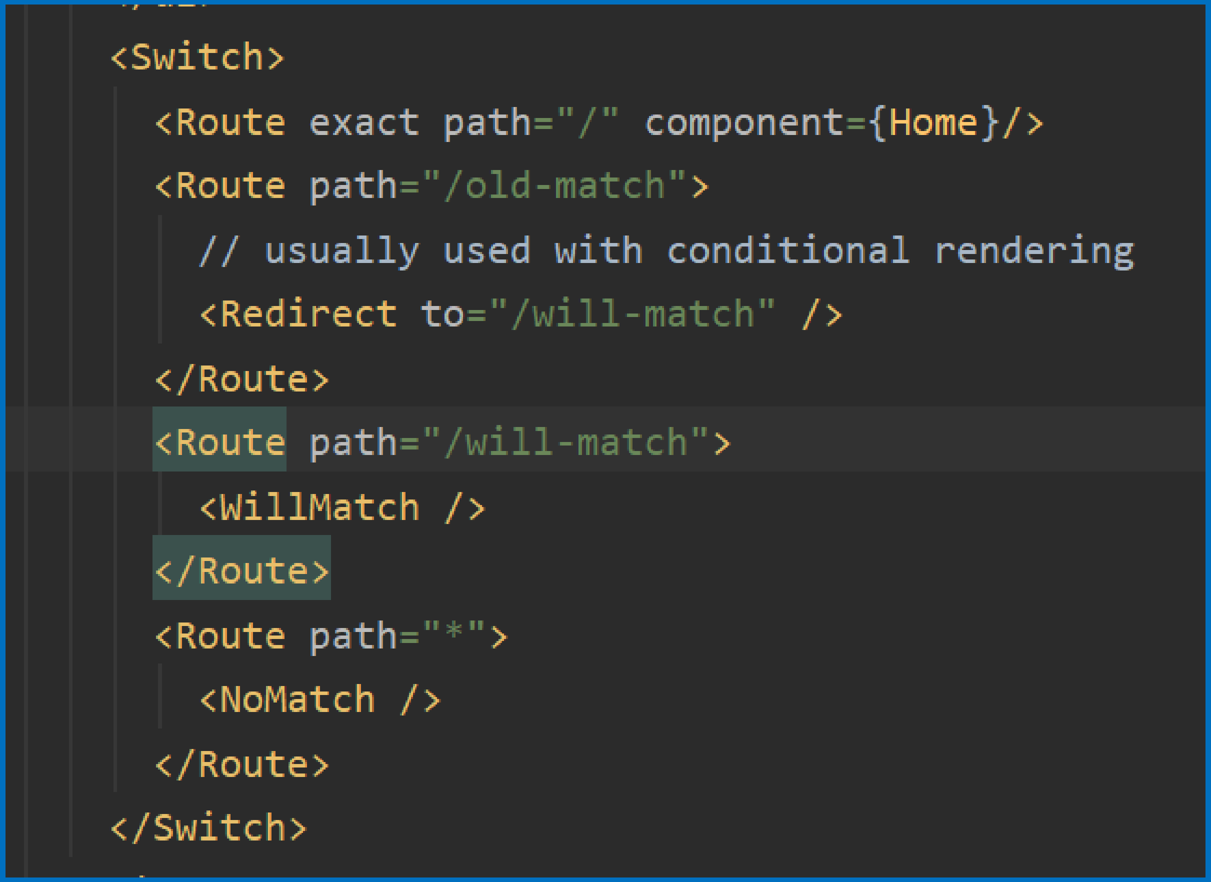
- Routing is used by placing components as children of the Router component, meaning inside Router-tags.



```
import {
  BrowserRouter as Router,
  Route,
  Link,
  Switch,
  Redirect,
  useLocation
} from 'react-router-dom';
```

Router

- First, everything needs to go inside BrowserRouter
- Next, add the Switch element. These ensure that only one component is rendered at a time.
- Then add the <Route> tags. These are the links between the components and should be placed inside the <Switch> tags.
- To tell the <Route> tags which component to load, simply add a path attribute and the name of the component you want to load with component attribute.

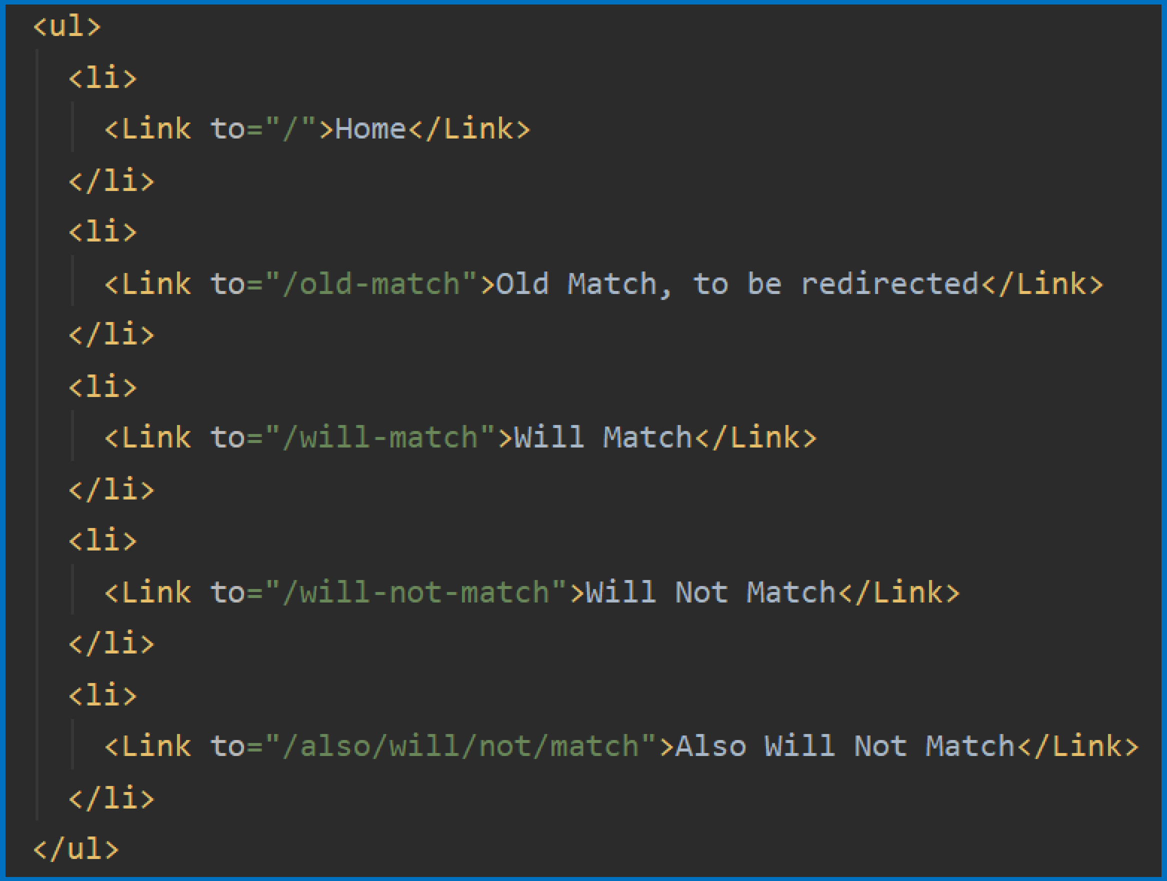


```
<Switch>
  <Route exact path="/" component={Home}/>
  <Route path="/old-match">
    // usually used with conditional rendering
    <Redirect to="/will-match" />
  </Route>
  <Route path="/will-match">
    <WillMatch />
  </Route>
  <Route path="*">
    <NoMatch />
  </Route>
</Switch>
```

Note: Order Matters! A <Switch> looks through its children <Route>s and renders the first one that matches the current URL.

Router

- The exact keyword will only match if the path matches the location.pathname exactly.
- To add clickable links to the site, we use the Link element from React Router



```
<ul>
  <li>
    <Link to="/">Home</Link>
  </li>
  <li>
    <Link to="/old-match">Old Match, to be redirected</Link>
  </li>
  <li>
    <Link to="/will-match">Will Match</Link>
  </li>
  <li>
    <Link to="/will-not-match">Will Not Match</Link>
  </li>
  <li>
    <Link to="/also/will/not/match">Also Will Not Match</Link>
  </li>
</ul>
```

Nested Routing

- Since routes are regular React components, they may be rendered anywhere in the app, including in child elements.
- This helps when it's time to code-split your app into multiple bundles because code-splitting a React Router app is the same as code-splitting any other React app.

Path Variable

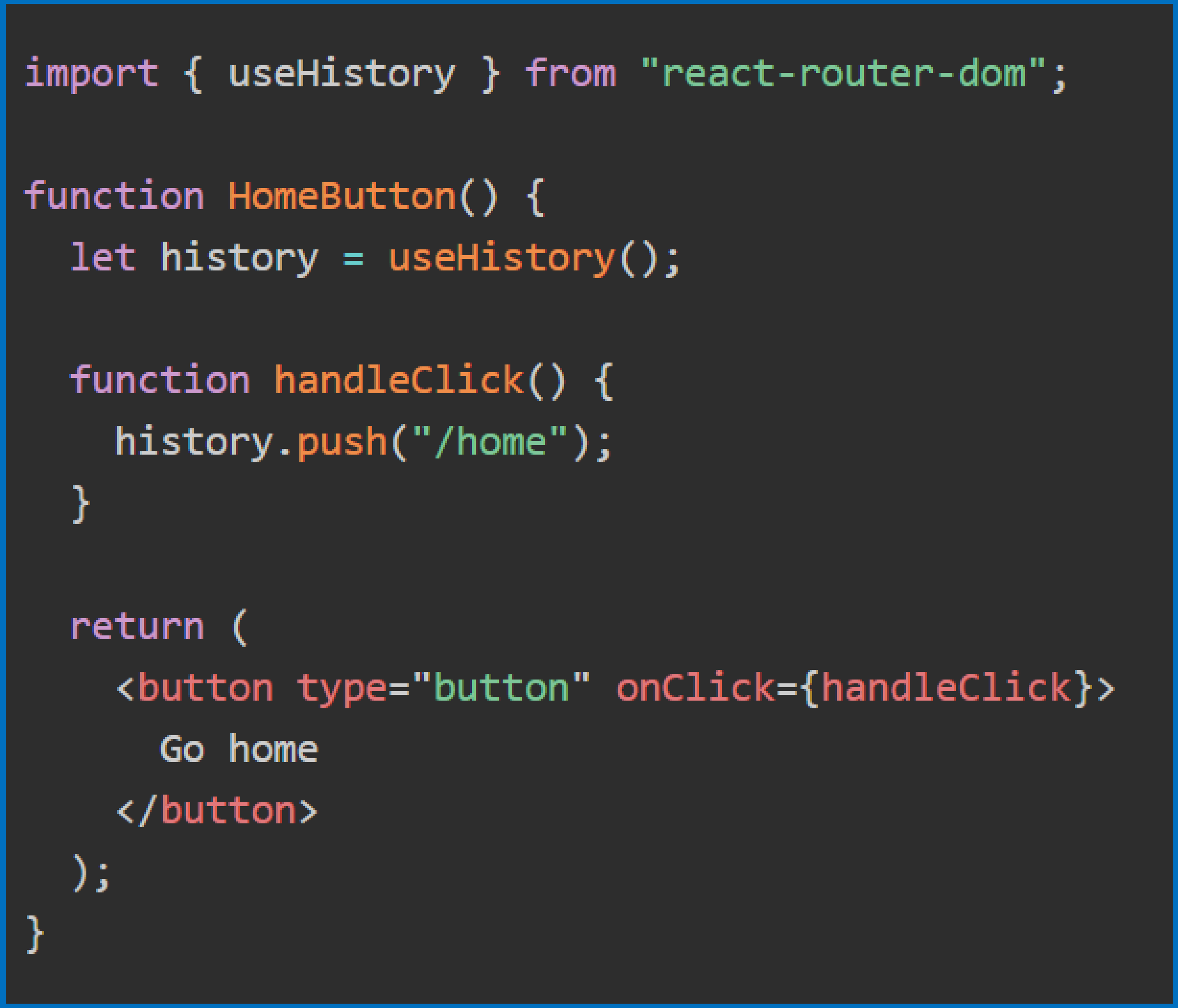
- Params are placeholders in the URL that begin with a colon, like the `:id` param defined in the route in this example.
- A similar convention is used for matching dynamic segments in other popular web frameworks like Rails and Express.
- We can use the `useParams` hook here to access the dynamic pieces of the URL.

QUERY PARAMETER

- React Router does not have any opinions about how you should parse URL query strings.
- If you use simple key=value query strings and you do not need to support IE 11, you can use the browser's built-in URLSearchParams API.
- If your query strings contain array or object syntax, you'll probably need to bring your own query parsing function.
- ***Example***

useHistory

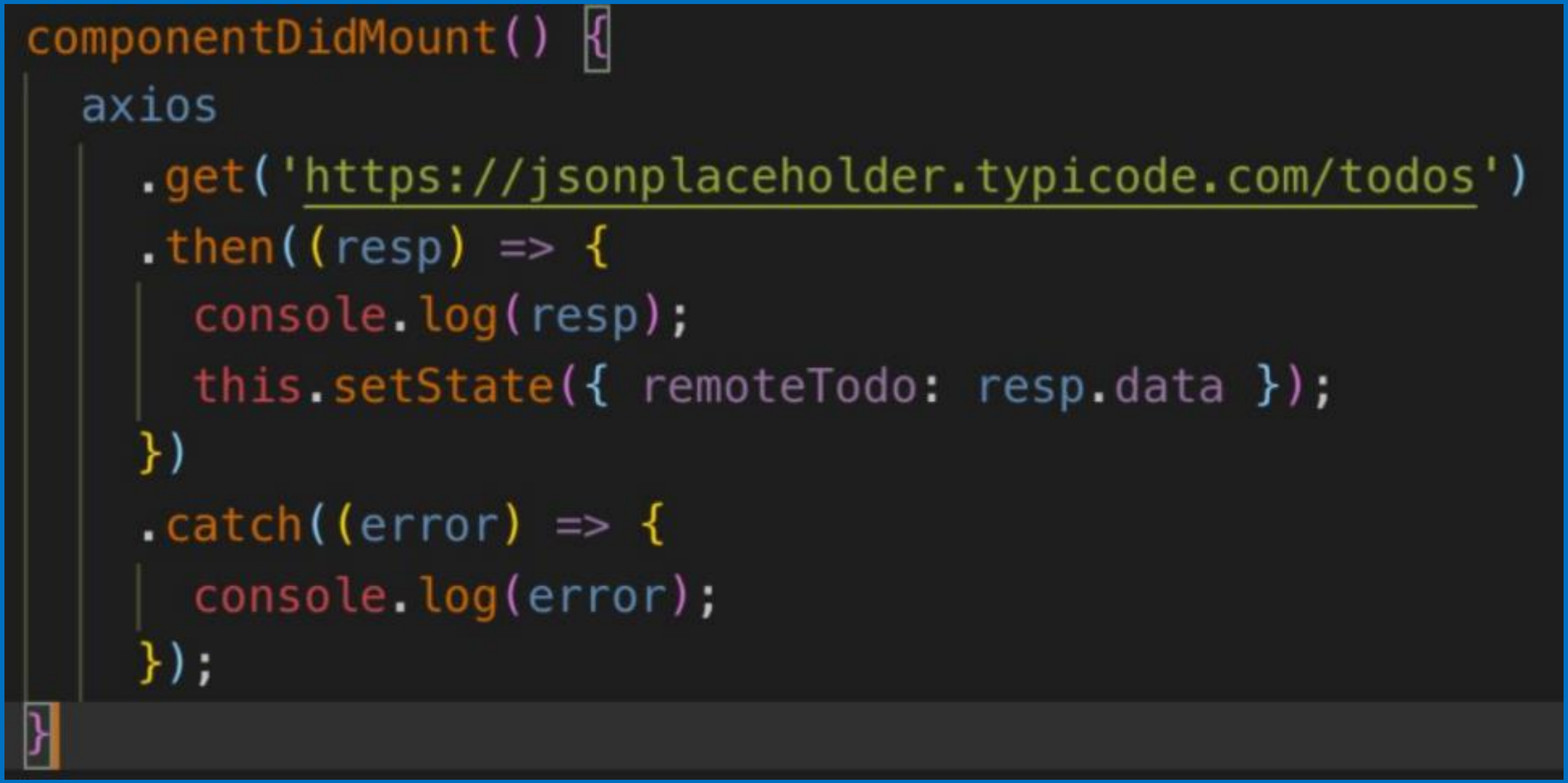
- The useHistory hook gives you access to the history instance that you may use to navigate



```
import { useHistory } from "react-router-dom";

function HomeButton() {
  let history = useHistory();

  function handleClick() {
    history.push("/home");
  }

  return (
    <button type="button" onClick={handleClick}>
      Go home
    </button>
  );
}
```

HTTP Request

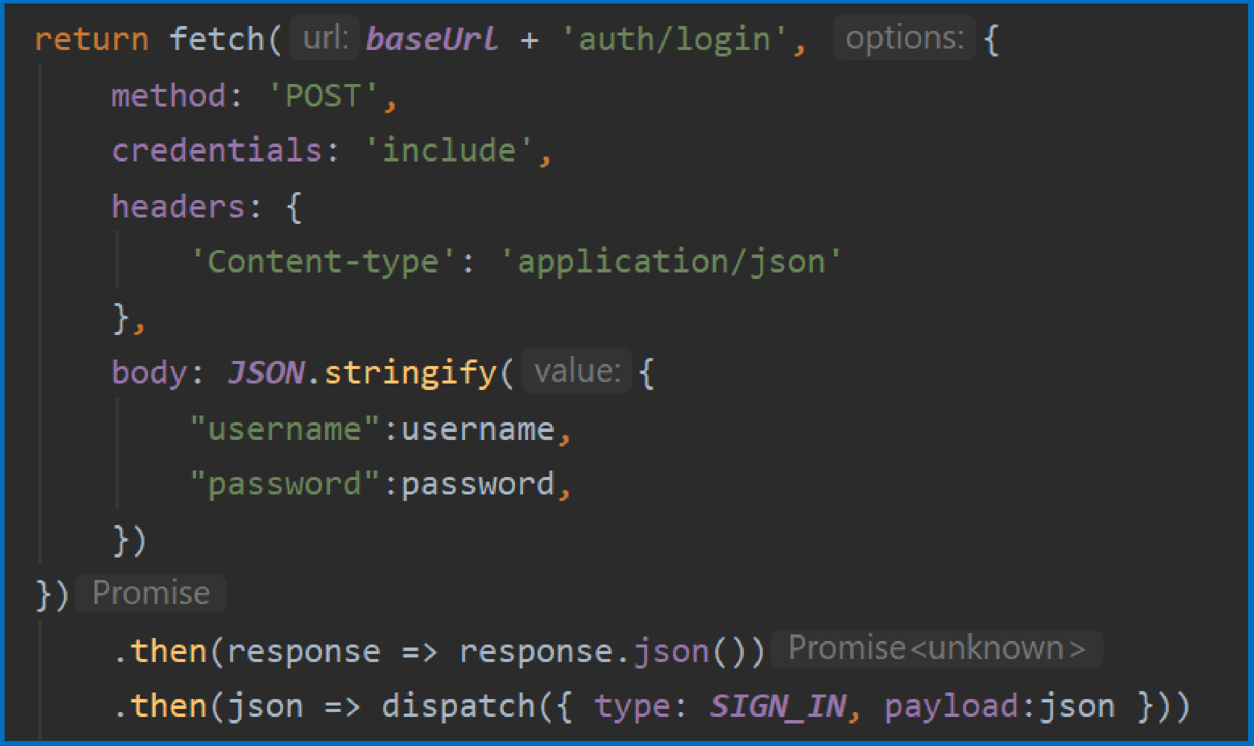
- ***axios*** is a Promise-based HTTP client for browsers and nodejs. It is essentially a encapsulation of native XHR, except that it is an implementation version of Promise and conforms to the latest ES specifications.



```
componentDidMount() {
  axios
    .get('https://jsonplaceholder.typicode.com/todos')
    .then((resp) => {
      console.log(resp);
      this.setState({ remoteTodo: resp.data });
    })
    .catch((error) => {
      console.log(error);
    });
}
```

HTTP Request

- The ***Fetch API*** provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.
- This kind of functionality was previously achieved using XMLHttpRequest. Fetch provides a better alternative that can be easily used by other technologies such as Service Workers. Fetch also provides a single logical place to define other HTTP-related concepts such as CORS and extensions to HTTP.



```
return fetch( url: baseUrl + 'auth/login', options: {
    method: 'POST',
    credentials: 'include',
    headers: {
        'Content-type': 'application/json'
    },
    body: JSON.stringify( value: {
        "username":username,
        "password":password,
    })
}) Promise
    .then(response => response.json()) Promise<unknown>
    .then(json => dispatch({ type: SIGN_IN, payload:json }))
```

Fetch API

- A fetch() promise will reject with a TypeError when a network error is encountered or CORS is misconfigured on the server-side, although this usually means permission issues or similar — a 404 does not constitute a network error, for example. An accurate check for a successful fetch() would include checking that the promise resolved, then checking that the Response.ok property has a value of true. The code would look something like this:

```
fetch('flowers.jpg')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.blob();
  })
  .then(myBlob => {
    myImage.src = URL.createObjectURL(myBlob);
  })
  .catch(error => {
    console.error('There has been a problem with your fetch operation:', error);
  });
```

Two Questions

1. How can we write async logic (We use axios or fetch to get data from backend, we will dealing with async logic in our application) that interacts with the store?

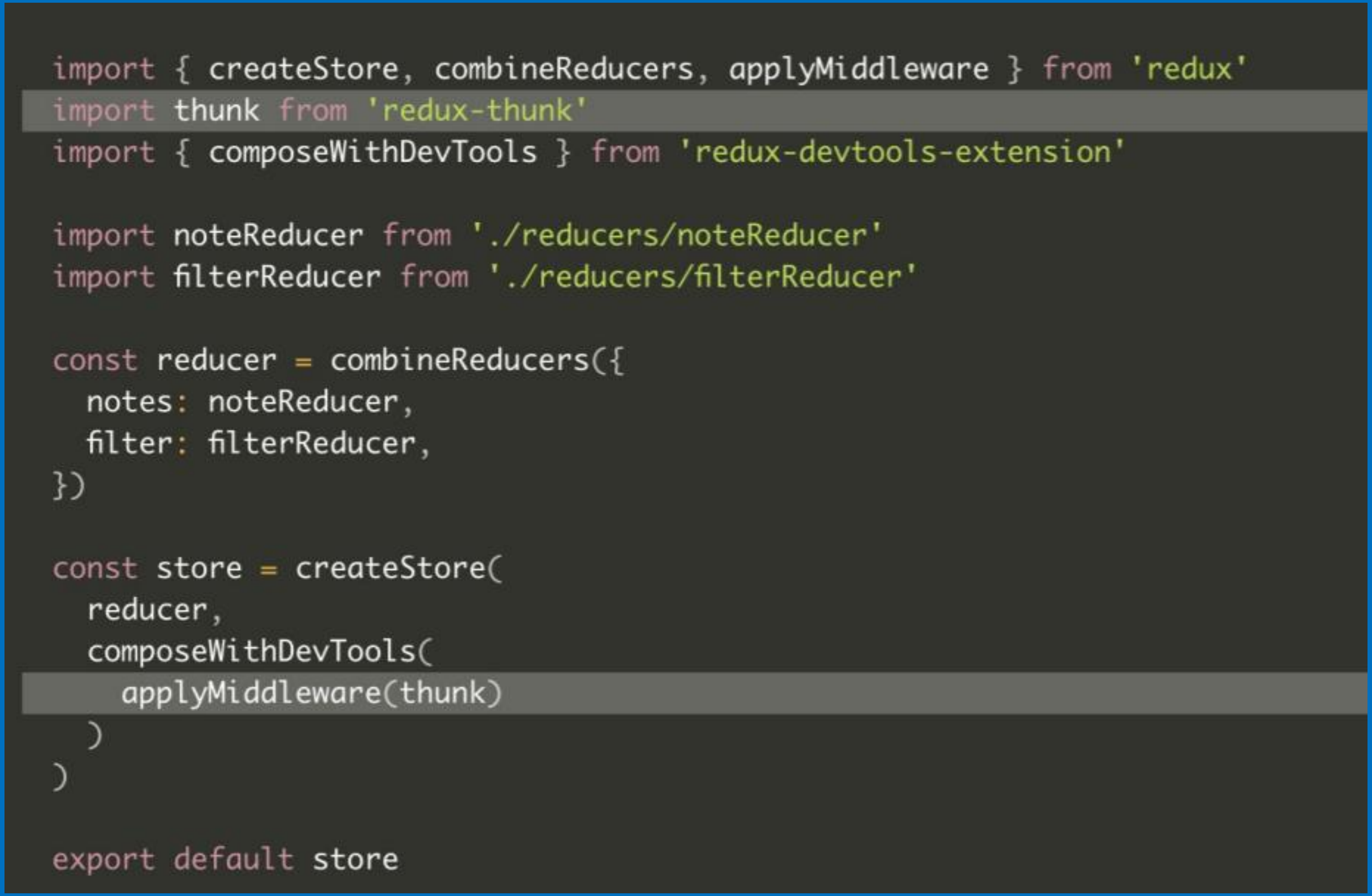
   With a plain basic Redux store, you can only do simple synchronous updates by dispatching an action. So we can take the advantage of the Middleware, Middleware extends the store's abilities, and lets you write async logic that interacts with the store.

1. Can we add router guard for react router like Angular?

   We can use the JSX if logic to “create” the guard by ourselves. Also, there are lots of libraries in the NPM, you can do some research.

Middleware - thunk

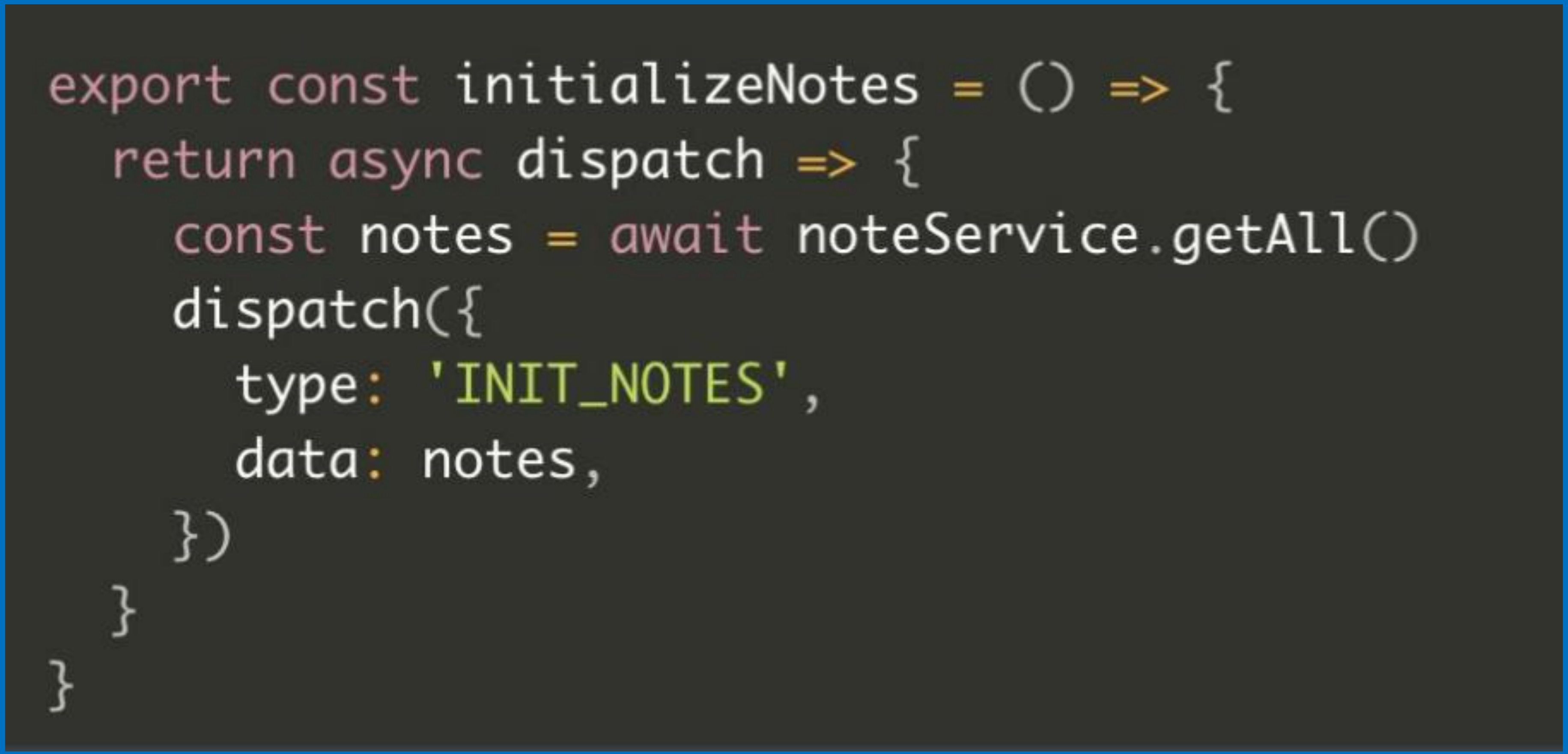
- Redux Thunk extends the store's abilities, and lets you write async logic that interacts with the store.
- It just helps us to create asynchronous actions.



```
import { createStore, combineReducers, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import { composeWithDevTools } from 'redux-devtools-extension'

import noteReducer from './reducers/noteReducer'
import filterReducer from './reducers/filterReducer'

const reducer = combineReducers({
  notes: noteReducer,
  filter: filterReducer,
})

const store = createStore(
  reducer,
  composeWithDevTools(
    applyMiddleware(thunk)
  )
)

export default store
```

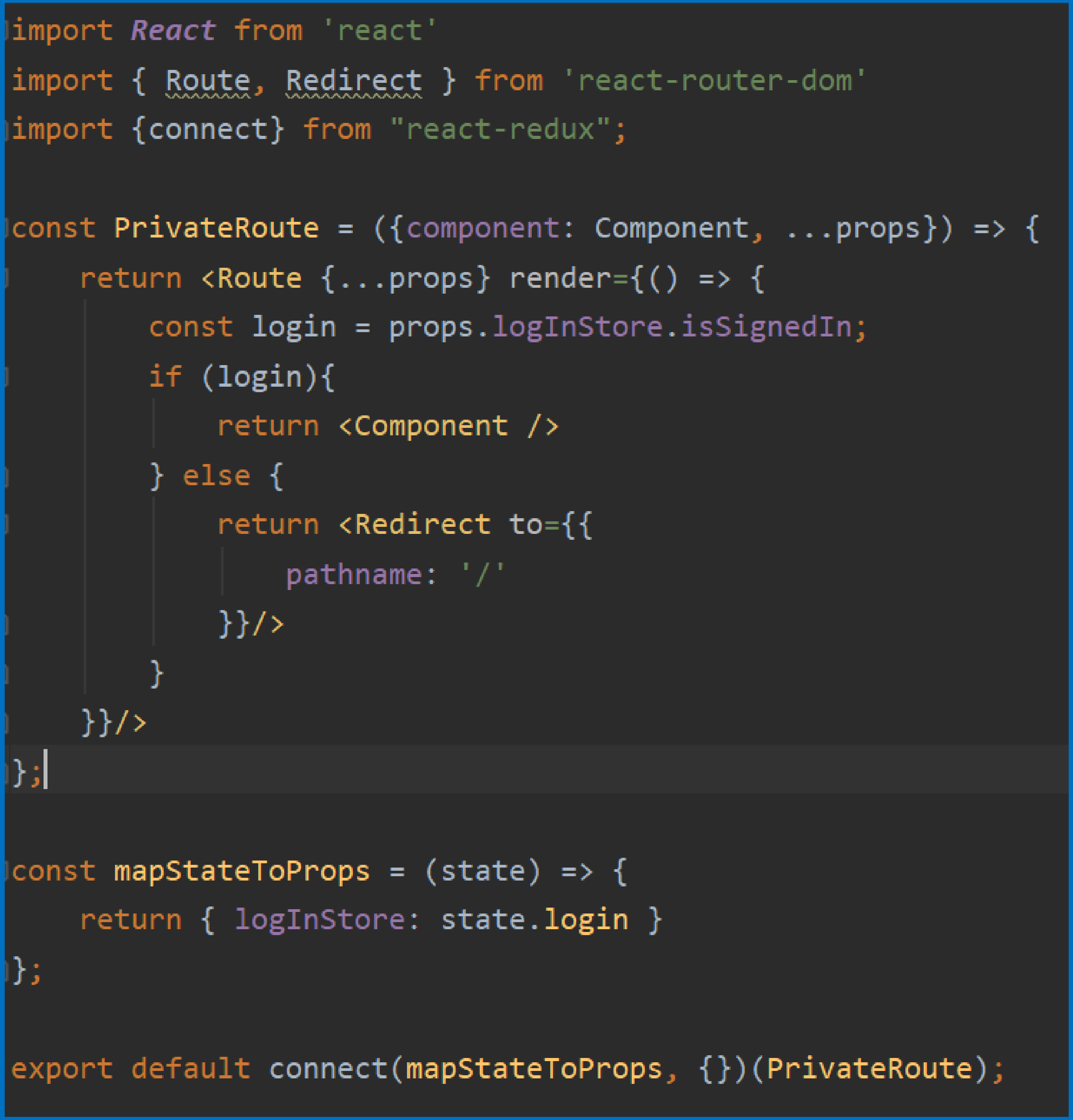
Middleware - thunk

- With asynchronous action creators, which first wait for some operation to finish, after which they then dispatch the real action.



```
export const initializeNotes = () => {
  return async dispatch => {
    const notes = await noteService.getAll()
    dispatch({
      type: 'INIT_NOTES',
      data: notes,
    })
  }
}
```

Add Guard Manually



```
import React from 'react'
import { Route, Redirect } from 'react-router-dom'
import {connect} from "react-redux";

const PrivateRoute = ({component: Component, ...props}) => {
    return <Route {...props} render={() => {
        const login = props.logInStore.isSignedIn;
        if (login){
            return <Component />
        } else {
            return <Redirect to={{
                pathname: '/'
            }}/>
        }
    }}/>
};

const mapStateToProps = (state) => {
    return { logInStore: state.login }
};

export default connect(mapStateToProps, {})(PrivateRoute);
```

ANY QUESTIONS?