📝 - The Redux Pattern

Not completed yet

What is Redux?

Redux is a predictable state container for JavaScript apps. It can be applied on many libraries and frameworks, but probably its most common implementation comes with ReactJS.

Its purpose it's about giving all your components a SHARED state they can access (apart from their internal ones) and to maintain a centralized single source-of-truth for the whole app, in a read-only form.

Each component can access this shared state from anywhere in your application, without the need to lifting states up all the times and drilling the props down just to reach inner components.

Immutability

This shared state is read-only and therefore cannot be modified. For updating it you need to replace it entirely with a new one: this process needs a specific set of steps to be performed, which we'll explore in a second.

Highly predictable

The Redux Store is also highly predictable, because any new version of the shared state is going to be mathematically computed from the previous one and the single modification you want to apply. There's no room for randomness or unreliability: each new iteration of the Redux Store will be atomic, reversible and pretty easy to compare with the previous one.

Companies ❤️ Redux

The immutable and predictable nature of the Redux Store received much appreciation by companies and enterprises, and nowadays its usage is very widespread on big and complex applications.
If you're going to work on big projects involving ReactJS, chances are you're also going to deal with Redux.

📝 - How does it work?

The Redux Store

The shared state hosted by Redux is a JS object, with a predetermined shape and an initial value. Here's an example:

```
{
  count: 0
}
```

This state can be accessed from any component in you application, and every component also gets the ability to create a new iteration of the shared state starting from the last one.
This state can hold as many properties as you want.

Actions

For updating the state from a component you need to dispatch an action:

```
{
  type: "INCREMENT",
  payload: 1
}
```

An action in Redux is another JS object, holding at least a type property. The type property will tell Redux how to create a new state accordingly with the update you want to bring into it.

Reducers

Once an action gets dispatched, it will be immediately intercepted by a reducer. A reducer is a function, more specifically a pure function (given the same input, it will always emit the same output).
The reducer is physically in charge to compute and generate the new state of the application (a new JS object) starting from the last one and the action it intercepted.
Here's a reducer example:

```
export const mainReducer = (state = {}, action) => {
  switch (action.type) {
    case "INCREMENT":
      return {
        ...state,
        count: state.count + action.payload
      };
    default:
      return state
  }
}
```

As you can see, the reducer will always return a new object (it will never alter the previous state) based on its parameters, the old state and the action caught.
The new state will have the same properties as the old one, plus the update you want to introduce.
The switch statement is very common for a reducer function because it's easy to read and will automatically provide a default case, which is very handy for not altering the state at all and straight return it if an unknown action is intercepted.

📝 - Connecting a component

The connect function

For making your React component capable of accessing the Redux shared state or dispatching actions in order to create new ones, you need to connect it.
Redux provides a convenient connect function, which will enrich your component with additional props coming from the Store.
The connect function works with two parameters: their default names are *mapStateToProps* and *mapDispatchToProps*.

```
export default connect(mapStateToProps, mapDispatchToProps)(App)
```

mapStateToProps

mapStateToProps is a function receiving the whole shared state as its parameter, and needs to return an object with the properties you want your component to receive as props.

```
const mapStateToProps = (state) => state
```

Declaring like this (returning the whole state) will bind all the properties of the shared state object as props for your component; with a shared state object declared like

```
{
  count: 0
}
```

you may expect your component to receive the *count* property from this.props.count.

mapDispatchToProps

mapDispatchToProps is a function receiving the dispatch method as parameter and needs to return an object with the methods you want your component to receive as props. Those methods can be used to dispatch actions with various types.
Example:

```
const mapDispatchToProps = (dispatch) => ({
  increment: () =>
    dispatch({
      type: 'INCREMENT',
    })
})
```

The method *increment* can be fired using this.props.increment(), and with a reducer function set up as the other examples should increment the count variable in the shared state by 1 every time it's invoked.

Slides:
https://docs.google.com/presentation/d/1FoMDWc-siEi2aw4nQ6uCVDGtHnqYe_RhZ429nI2y63g/edit#slide=id.gaff26062cf_0_0

CodePill: https://youtu.be/n93GQorqcMY