📝 - The Redux Pattern

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

📝 - Multiple Reducers

Once the logic of your frontend becomes too messy and verbose for a single reducer, chances are you want to re-organize its structure in smaller chunks.
combineReducers is a function coming straight from the redux library which allows you to write independent and separate reducers, each one in charge of updating just a slice of the shared state object.
Once you split your logic into separate and smaller reducers, you can use the combineReducers function to join them and feed the createStore function of redux.
The keys in the object passed as the parameter of combineReducers must match the ones you are using in the initial state; so, with an object declared like this:

```
export const initialState = {
  cart: {
    products: [],
  },
  user: {
    firstName: "",
  },
};
```

and two reducers, one for the cart slice of it and one for the user, you want to pass keys that match:

```
const bigReducer = combineReducers({
  cart: cartReducer,
  user: userReducer
});
```

In this way the cartReducer will be in charge of updating just the cart slice of the state object, while the userReducer will take care of the user slice.
Because of this, probably you'll need to tweak the logic of your reducers: their initial state will be just a part of the original one, and the properties to return are going to be just a portion of their previous one.

```
const cartReducer = (state = initialState.cart, action) => {
  switch (action.type) {
    case "ADD_ITEM_TO_CART":
      return {
        ...state,
        products: [...state.products, action.payload],
```

```
      };
    case "REMOVE_ITEM_FROM_CART":
      return {
        ...state,
          products: state.products.filter((p, i) => i !== action.payload),
      };
    default:
      return state;
  }
};
```

📝 - redux-thunk

redux-thunk is a pretty popular middleware for the Redux library. It's main goal is to provide the user the ability to perform asynchronous operations in the Redux workflow (or to delay gracefully the dispatching of an action).
By definition, action dispatching and handling in Redux are synchronous operations: this happens by design, because of the predictable nature of the pattern.
But can we use the Redux Store for holding data coming from an async operation, like a fetch() ? And if we can, what is the best place for putting this asynchronous logic?

The reducers are pure functions, and therefore they cannot perform any side-effect or remote call in their code. They just compute new states with mathematical operations.
The only place left in the pattern is the action creator, which typically is just a function that returns a Redux action (a JS object).

This is an action creator:

```
export const addToCartAction = (book) => ({
  type: 'ADD_ITEM_TO_CART',
  payload: book,
})
```

Once redux-thunk is injected into the Redux flow, action creators suddenly become capable of also dispatching functions instead of objects:

```
export const addToCartActionWithThunk = (book) => {
  return (dispatch, getState) => {
```

```
    console.log("A thunk was used to dispatch this action", getState());
    // you can also put async logic here!
    dispatch({
      type: "ADD_ITEM_TO_CART",
      payload: book,
    });
  };
};
```

As you can see the function that now can be returned from an action creator receives from Redux the dispatch and the getState methods (this last one is handy for checking the current state in any moment during you async operations).

Now you can safely delay the dispatching of an action waiting for an asynchronous operation to complete or for performing any data manipulation beforehand, without messing up the predictable nature of your reducers (which will get involved just when the data is ready).

Without a middleware like this, Redux will just accept the dispatching of normal JS objects; now it will still accepts normal actions, but if a function is found it will enrich it with the dispatch and getState methods.

📝 - TypeScript Intro & Setup

TypeScript is an open-source programming language built upon JavaScript.

It is called a "super-set" of the JS Language, because it enriches it with new features like static typing and smart error checking; once the development process is complete, it compiles back into plain JS to be digestible to any web browser and NodeJS environment.

It is especially useful on large-scaled applications, because using it you'll avoid losing track on how your data is shaped and you'll be "forced" to use good practices and write cleaner code!

TypeScript Setup

I'll suggest you to install TypeScript via npm, using the -g flag:

```
npm install -g typescript
```

This will make sure TypeScript gets installed *globally* on your system, allowing you to use the tsc command in any project.

The TypeScript compiler (tsc)

Once you create your TypeScript files (with a .ts file extension) you can compile them back into regular JS invoking the Typescript compiler through the command line:

```
tsc <put-file-path-here>
```

This will generate a .js file with the same name, containing the transpiled code. This file is now ready to be interpreted by the browser or the NodeJS engine.

If you want to automatically generate an updated .js version of your code at every file save, you can leave the tsc process running providing a watch parameter to your command:

```
tsc <put-file-path-here> -w
```