Getting back to the Refactoring Process

In order to refactor our React app, we will first create the reducer directory along with our first primitive reducer function. We must remember that the reducer always returns a new state of the app.

Let's get back to refactoring the Hello World React application to use Redux.

If I lost you at any point in the previous section, please read the section just one more time and I'm sure it'll sink. Better still, you can ask me a question.

Okay so here's all the code we have at this point.

```
import React, { Component } from "react";
import HelloWorld from "./HelloWorld";
import { createStore } from "redux";
const store = createStore(reducer);

class App extends Component {
  render() {
    return
  }
}

export default App;
```

Makes sense?

You may have noticed a problem with this code. See Line 4.

The `reducer` function passed into `createStore` doesn't exist yet.

Now we need to write one. The reducer is just a function, remember?

Create a new directory called reducers, and create an `index.js` file in there. Essentially, our reducer function will be in the path, `src/reducers/index.js`

First export a simple function in this file:

```
export default () => {

}
```

Remember, that the reducer takes in two arguments - as established earlier. Right now, we'll concern ourselves with the first argument, STATE.

Put that into the function, and we have this:

```
export default (state) => {
}
```

Not bad.

A reducer always returns something. In the initial `Array.reduce()` reducer example, we returned the sum of the accumulator and current value.

For a Redux Reducer, you always return the NEW STATE of your application.

Let me explain.

After you walk into the bank and make a successful withdrawal, the current amount of money held in the bank's vault for you is no longer the same. Now, if you withdrew $200, you are now short of $200 i.e You have an account balance of minus $200.

Again, the Cashier and Vault remain in sync on how much you now have.

Just like the Cashier, this is exactly how the Reducer works.

Like the Cashier, the Reducer always returns the new state of your application. Just in case something has changed. We don't want to issue the same bank balance even though a withdrawal action was performed.

We'll get to the internals of changing/updating the state later on. For now, blind trust will suffice. Now, back to the problem at hand.

Since we aren't concerned about changing/updating the state at this point, we will keep NEW STATE being returned as the same state passed in.

Here's the representation of this within the reducer:

```
export default (state) => {
  return state
}
```

If you go to the bank without performing an action, your bank balance remains the same, right?

Since we aren't performing any ACTION or even passing that into the reducer yet, we will just return the same state.

With our reducer created, let's step back for a moment. Remember when we said that the Store acts as the Vault which contains the state object? Well, it is time to create that vault.