📝 - 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.