05. REDUX Store.

In the previous sections, we defined the **actions** that represent the facts about "what happened" and the **reducers** that update the state according to those actions.

5.1. Store

The **Store** is the object that brings them (actions and reducers) together. The store has the following responsibilities:

- Holds application state;
- Allows access to state via **getState**();
- Allows state to be updated via **dispatch**(action);
- Registers listeners via **subscribe**(listener);
- Handles unregistering of listeners via the function returned by **subscribe**(listener).

It's important to note that you'll only have a single store in a Redux application. When you want to split your data handling logic, you'll use **reducer composition** instead of many stores.

https://redux.js.org/basics/reducers#splitting-reducers

It's easy to create a store if you have a reducer. In the previous section, we used **combineReducers**() to combine several reducers into one. We will now import it, and pass it to **createStore**().

Example:

```
import { createStore } from 'redux'
import todoApp from './reducers'

const store = createStore(todoApp)
```

You may optionally specify the initial state as the second argument to createStore(). This is useful for hydrating the state of the client to match the state of a Redux application running on the server.

Example:

```
const store = createStore(todoApp, window.STATE_FROM_SERVER)
```

5.2. Dispatching Actions

Now that we have created a store, let's verify our program works! Even without any UI, we can already test the update logic.

Example:

```
import {
  addTodo,
  toggleTodo,
  setVisibilityFilter,
  VisibilityFilters
} from './actions'


// Log the initial state

console.log(store.getState())


// Every time the state changes, log it
// subscribe() returns a function for unregistering the listener

const unsubscribe = store.subscribe(() =>
console.log(store.getState()))



// Dispatch some actions

store.dispatch(addTodo('Learn about actions'))
store.dispatch(addTodo('Learn about reducers'))
store.dispatch(addTodo('Learn about store'))
store.dispatch(toggleTodo(0))
store.dispatch(toggleTodo(1))
store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED))



// Stop listening to state updates

unsubscribe()
```

You can see how this causes the state held by the store to change:

Output:

```
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[0]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[1]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[2]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▼ Object {visibleTodoFilter: "SHOW_COMPLETED", todos: Array[3]} ⓘ
  ▼ todos: Array[3]
    ▼ 0: Object
        completed: true
        text: "Learn about actions"
      ▶ __proto__: Object
    ▼ 1: Object
        completed: true
        text: "Learn about reducers"
      ▶ __proto__: Object
    ▼ 2: Object
        completed: false
        text: "Learn about store"
      ▶ __proto__: Object
      length: 3
    ▶ __proto__: Array[0]
    visibleTodoFilter: "SHOW_COMPLETED"
  ▶ __proto__: Object
```

We specified the behavior of our app before we even started writing the UI. We won't do this in this tutorial, but at this point you can write tests for your reducers and action creators. You won't need to mock anything because **they are just pure functions**. Call them, and make assertions on what they return.

5.3. Source Code.

Example: index.js

```
import { createStore } from 'redux'
import todoApp from './reducers'

const store = createStore(todoApp)
```