00. REDUX Introduction.

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time travelling debugger.

You can use Redux together with React, or with any other view library.

It is tiny (2kB, including dependencies), but has a large ecosystem of add-ons available.

0.1. Installation

Redux is available as a package on **NPM** for use with a module bundler or in a Node application:

Example:

```
npm install --save redux
```

It is also available as a precompiled UMD package that defines a ***window.Redux*** global variable. The UMD package can be used as a *<script>* tag directly.

Example:

```
<script src="https://unpkg.com/redux/dist/redux.js"></script>
```

For more details, see the Installation page: https://redux.js.org/introduction/installation

0.2. Redux Starter Kit

Redux itself is small and unopinionated. We also have a separate package called **redux-starter-kit**, which includes some opinionated defaults that help you use Redux more effectively.

https://redux-starter-kit.js.org/

It helps simplify a lot of common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "**slices**" of state at once.

Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, **redux-starter-kit** can help you make your Redux code better.

0.3. Basic Example

The whole state of your app is stored in an object tree inside a single store.
The only way to change the state tree is to emit an **action**, an object describing what happened.
To specify how the actions transform the state tree, you write pure **reducers**. That's it!

Example:

```
import { createStore } from 'redux'

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

let store = createStore(counter)

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

store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'DECREMENT' })
```

Explanation: This is a reducer, a pure function with (state, action) => state signature.

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

It describes how an action transforms the state into the next state. The shape of the state is up to you: it can be a primitive, an array, an object, or even an Immutable.js data structure. The only important part is that you should not mutate the state object, but return a new object if the state changes.

In this example, we use a `switch` statement and strings, but you can use a helper that follows a different convention (such as function maps) if it makes sense for your project.

Example: Create a Redux store holding the state of your app. Its API is { subscribe, dispatch, getState }.

```
let store = createStore(counter)
```

Example: You can use subscribe() to update the UI in response to state changes.

```
store.subscribe(() => console.log(store.getState()))
```

Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly. However it can also be handy to persist the current state in the localStorage.

Example: The only way to mutate the internal state is to dispatch an action. The actions can be serialized, logged or stored and later replayed.

```
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'DECREMENT' })
```

Instead of mutating the state directly, you specify the mutations you want to happen with plain objects called **actions**. Then you write a special function called a **reducer** to decide how every action transforms the entire application's state.

In a typical Redux app, **there is just a single store with a single root reducing function.** As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.

This architecture might seem like an overkill for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused it. You can record user sessions and reproduce them just by replaying every action.

0.4. Motivation

As the requirements for JavaScript single-page applications have become increasingly complicated, **our code must manage more state than ever before.** This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on.

Managing this ever-changing state is hard. If a model can update another model, then a view can update a model, which updates another model, and this, in turn, might cause another view to update. At some point, you no longer understand what happens in your app as you have **lost control over the when, why, and how of its state.** When a system is opaque and non-deterministic, it's hard to reproduce bugs or add new features.

As if this weren't bad enough, consider the **new requirements becoming common in front-end product development.** As developers, we are expected to handle optimistic updates, server-side rendering, fetching data before performing route transitions, and so on. We find ourselves trying to manage a complexity that we have never had to deal with before, and we inevitably ask the question: is it time to give up? The answer is no.

This complexity is difficult to handle as **we're mixing two concepts** that are very hard for the human mind to reason about: **mutation and asynchronicity**. I call them Mentos and Coke. Both can be great in separation, but together they create a mess. Libraries like React attempt to solve this problem in the view layer by removing both asynchrony and direct DOM manipulation. However, managing the state of your data is left up to you. This is where Redux enters.

Following in the steps of Flux, CQRS, and Event Sourcing, **Redux attempts to make state mutations predictable** by imposing certain restrictions on how and when updates can happen. These restrictions are reflected in the **three principles of Redux**.

- Flux: http://facebook.github.io/flux
- CQRS: http://martinfowler.com/bliki/CQRS.html
- Event Sourcing: http://martinfowler.com/eaaDev/EventSourcing.html