01. REDUX Core Concepts.

Redux is a predictable state container for JavaScript apps, and a very valuable tool for organizing application state. It's a popular library to manage state in React apps, but it can be used just as well with Angular, Vue.js or just plain old vanilla JavaScript.

One thing most people find difficult about Redux is knowing when to use it. The bigger and more complex your app gets, the more likely it's going to be that you'd benefit from using Redux. If you're starting to work on an app and you anticipate that it'll grow substantially, it can be a good idea to start out with Redux right off the bat so that as your app changes and scales you can easily implement those changes without refactoring a lot of your existing code.

In this brief introduction to Redux, we'll go over the main concepts:

- **reducers,**
- **actions,**
- **action creators** and
- **store.**

It can seem like a complex topic at first glance, but the core concepts are actually pretty straightforward.

1.1. Core Concepts

Imagine your app's state is described as a plain object. For example, the state of a todo app might look like this:

Example:

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

This object is like a "model" except that there are no setters. This is so that different parts of the code can't change the state arbitrarily, causing hard-to-reproduce bugs.

To change something in the state, you need to dispatch an action. An action is a plain JavaScript

object (notice how we don't introduce any magic?) that describes what happened. Here are a few example actions:

Example:

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Enforcing that every change is described as an action lets us have a clear understanding of what's going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened. Finally, to tie state and actions together, we write a function called a reducer. Again, nothing magical about it—it's just a function that takes state and action as arguments, and returns the next state of the app. It would be hard to write such a function for a big app, so we write smaller functions managing parts of the state:

Example:

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter
  } else {
    return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map((todo, index) =>
        action.index === index
          ? { text: todo.text, completed: !todo.completed }
          : todo
      )
    default:
      return state
  }
}
```

And we write another reducer that manages the complete state of our app by calling those two reducers for the corresponding state keys:

Example:

```
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter: visibilityFilter(state.visibilityFilter,
action)
  }
}
```

This is basically the whole idea of Redux. Note that we haven't used any Redux APIs. It comes with a few utilities to facilitate this pattern, but the main idea is that you describe how your state is updated over time in response to action objects, and 90% of the code you write is just plain JavaScript, with no use of Redux itself, its APIs, or any magic.

1.2. What's a Reducer?

A reducer is a pure function that takes the previous state and an action as arguments and returns a new state. Actions are an object with a type and an optional payload:

Example:

```
function myReducer(previousState, action) => {
  // use the action type and payload to create a new state based on
  // the previous state.
  return newState;
}
```

Reducers specify how the application's state changes in response to actions that are dispatched to the store.

Since **reducers are pure functions**, we don't mutate the arguments given to it, perform API calls, routing transitions or call non-pure functions like *Math.random() or Date.now()*.

If your app has multiple pieces of state, then you can have multiple reducers. For example, each major feature inside your app can have its own reducer. **Reducers are concerned only with the value of the state.**

1.3. What's an Action?

Actions are plain JavaScript objects that represent payloads of information that send data from your application to your store. Actions have a type and an optional payload.

Most changes in an application that uses Redux start off with an event that is triggered by a user either directly or indirectly. Events such as clicking on a button, selecting an item from a dropdown menu, hovering on a particular element or an AJAX request that just returned some data. Even the initial loading of a page can be an occasion to **dispatch** an action. Actions are often dispatched using an **action creator**.

Example:

```
{
    type: 'ADD_TODO',
    payload: {
      task,
      completed: false

    }
}
```

1.4. What's an Action Creator?

In Redux, an action creator is a function that returns an action object. Action creators can seem like a superfluous step, but they make things more portable and easy to test. The action object returned from an action creator is sent to all of the different reducers in the app.

Depending on what the action is, reducers can choose to return a new version of their piece of state. The newly returned piece of state then gets piped into the application state, which then gets piped back into our React app, which then causes all of our components to re-render.

So lets say a user clicks on a button, we then call an action creator which is a function that returns an action. That action has a type that describes the type of action that was just triggered.

Example: Here's an example action creator:

```
export function addTodo({ task }) {
  return {
    type: 'ADD_TODO',
    payload: {
      task,
```

```
        completed: false
      },
    }
}

// example returned value:
// {
//   type: 'ADD_TODO',
//   todo: { task: '🛒 get some milk', completed: false },
// }
```

And here's a simple reducer that deals with the action of type ADD_TODO:

Example:

```
export default function(state = initialState, action) {
  switch (action.type) {
    case 'ADD_TODO':
      const newState = [...state, action.payload];
      return newState;

    // Deal with more cases like 'TOGGLE_TODO', 'DELETE_TODO',...

    default:
      return state;
  }
}
```

All the reducers processed the action. Reducers that are not interested in this specific action type just return the same state, and reducers that are interested return a new state. Now all of the components are notified of the changes to the state. Once notified, all of the components will re render with new props:

Example:

```
{
  currentTask: { task: '🛒 get some milk', completed: false },
  todos: [
    { task: '🛒 get some milk', completed: false },
    { task: '🎷 Practice saxophone', completed: true }
  ],
}
```

1.5. Combining Reducers

Redux gives us a function called **combineReducers** that performs to tasks:

- It generates a function that calls our reducers with the slice of state selected according to their key.
- It then it combines the results into a single object once again.

1.6. What is the Store?

We keep mentioning the elusive store but we have yet to talk about what the store actually is.

In Redux, the store refers to the object that brings actions (that represent what happened) and reducers (that update the state according to those actions) together. **There is only a single store in a Redux application.**

The store has several duties:

- Allow access to state via getState().
- Allow state to be updated via dispatch(action).
- Holds the whole application state.
- Registers listeners using subscribe(listener).
- Unregisters listeners via the function returned by subscribe(listener).

Basically all we need in order to create a store are reducers. We mentionned **combineReducers** to combine several reducers into one. Now, to create a store, we will import combineReducers and pass it to **createStore**:

Example:

```
import { createStore } from 'redux';
import todoReducer from './reducers';

const store = createStore(todoReducer);
```

Then, we dispatch actions in our app using the store's dispatch method like so:

Example:

```
store.dispatch(addTodo({ task: '📖 Read about Redux'}));
store.dispatch(addTodo({ task: '🤔 Think about meaning of life' }));
// ...
```

1.7. Data Flow in Redux

One of the many benefits of Redux is that all data in an application follows the same lifecycle pattern. The logic of your app is more predictable and easier to understand, because Redux architecture follows a strict unidirectional data flow.

The 4 Main Steps of the Data Lifecycle in Redux:

- An event inside your app triggers a call to **store.dispatch(actionCreator(payload)).**
- The Redux store calls the root **reducer** with the current state and the action.
- The root reducer combines the output of multiple reducers into a single state tree.

Example:

```
export default const currentTask(state = {}, action){
  // deal with this piece of state
  return newState;
};

export default const todos(state = [], action){
  // deal with this piece of state
  return newState;
};

export default const todoApp = combineReducers({
  todos,
  currentTask,
});
```

When an action is emitted todoApp will call both reducers and combine both sets of results into a single state tree:

Example:

```
return {
  todos: nextTodos,
  currentTask: nextCurrentTask,
};
```

- The Redux store saves the complete state tree returned by the root reducer. The new state tree is now the **nextState** of your app.