Redux in Angular

# Redux

* Redux is a library that helps you manage the state of your application and is something that you should use in medium to large single page applications with complex data flows.
* If you're building a simple application with simple data flow you don't necessarily need redux. And in fact redux can add extra unnecessary complexity to your application.
* Without Redux, the state can be updated from anywhere which makes the application state unpredictable.
* Adding a new feature becomes a challenge as we don’t know the impact of this new feature on the Application State.
* Facebook had this problem back in 2014 and that's why they introduced the **Flux** architecture. Now **Redux** is a simplified and lightweight implementation of this architecture that provides a clean and elegant solution to this problem maintaining the application state in a predictable way.
* Redux is heavily based on functional programming.

## Benefits of Redux

* Provides a clean and elegant solution to this problem maintaining the application state in a **predictable** way.
* It **decouples** your application from a presentation framework like angular.
* So we can implement a big chunk of our application and its presentation logic using simple functions that are completely decoupled from Angular or any other presentation frameworks like React.
* It makes it easier to unit test your application without Mock's, spies and any other tricks that can make testing both complex and error prone.
* You can get some really cool tools as part of your development. One such example is redux developer tools extension that he can add to Chrome Firefox and other browsers. It makes it incredibly easy to debug your application by allowing you to inspect the application state in such a way that we have never seen before.
* Redux makes it incredibly easy to implement features like undo and redo. So if you need these functions in your application, you can implement it without redux but using redux makes your life much easier.

## When Redux should NOT be used

* All the fancy benefits of Redux of course they come with a cost just like many other architectural patterns.
* You're going to write it with more code and you'll have more moving parts in your application.
* So use redux only if you're building a medium to large single page application with complex views and data flows.

## When to use Redux

* When you have independent copies of the same data in multiple places
* When you have multiple views that need to work with the same data and be in sync.
* When Users can collaborate and work on the same piece of data so data can be changed by user A and at the same time user B can change the same data
* When data can be updated by multiple actors so it can be changed as a result of user actions. And at the same time it may arrive from the server either of our polling or push notifications.
* Remember when building a new app you don't need to automatically reach for redux. You can always start simple and as your application grows then you can refactor existing components and use redux to manage the application state in a predictable and deterministic way.

# Building Blocks of Redux

* Redux has 2 building blocks – The State, Actions and Reducers.

## State

* The store is a single JavaScript object that contains the state of the application.
* You can think of it as a local client side database.
* Different views and components use different parts or different slices of the application state depending on their functionality.
* This also means that if different components need to work with the same slice there is only one copy of that slice throughout the application.
* So once a component modifies that slice the changes are immediately visible to other components. We don't have multiple independent copies.
* It doesn’t consume too much memory unless you're storing 10000 or more objects in the store. But then why would you do that anyway.
* So for the most part it's OK to have a single object to store the entire application state.

## Actions

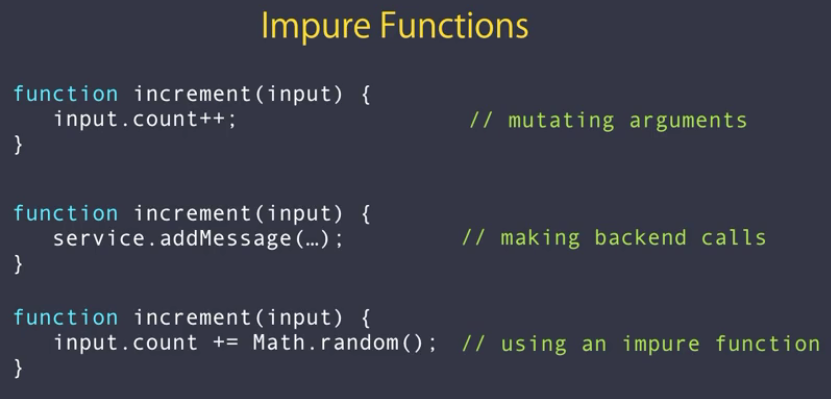
* Actions are plain JavaScript objects that represent something that has happened in the application.
* So semantically there are more like events.
* The difference between commands and events is **commands** represent something that **should happen** by posting a message whereas **events** indicate that something **has happened** e.g. A message was posted.
* Some example of Redux Actions –
  + When the user reads a message, we can represent this event using an action like this. {type: ‘MARK\_AS\_READ’}
  + As another example if the user posts a message we can have an action like this. {type: ‘POST\_MESSAGE’, body: ‘…’}
* So Actions are simple data structures and don't have any logic.
* By convention, we use a **type** property to specify the type of each action. An action can have additional properties like payload/body related to the event that just happened.

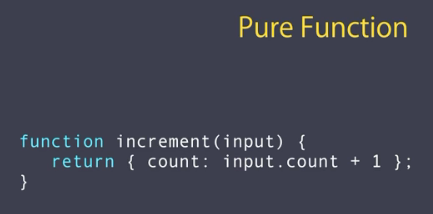
## Reducer

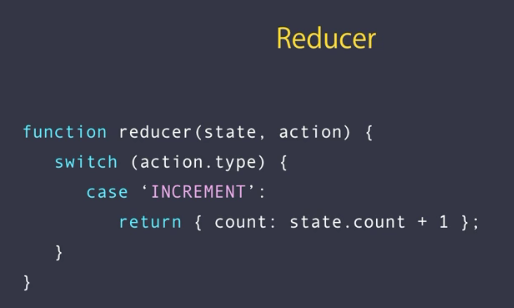
* Reducer is basically a function that specifies how the state changes in response to an action.
* You can think of it as an action handler or an event handler that determines how the state is changed.
* What is critical here is that the reducer does NOT modify the state. It only returns the new state and then the store will internally update the state.
* So nowhere in the application you're going to directly modify the state. This is the responsibility of the store. It keeps this up-to-date and updates it whenever necessary.
* In Redux, our reducers should be pure functions.

## Pure Function

* A function is pure if function is pure if you give it with the same input, we always get the same output no matter how many times we call that function.
* So it shouldn't have any side effects.
* In a pure function we should not mutate or modify any of the arguments.
* Some examples of Impure functions –



* Converting above impure functions to Pure functions –   
  
* Converting above Pure function to Redux Reducer –
  + Reducers are functions always take two arguments - the current state and an action.
  + Then is on the action type, they return a new state so typically we use a switch statement on the type property of the action and based on the value of the type property we return a new state.



## Benefits of having Reducers as Pure functions

* By looking at above example, you might say why we are not allowed to mutate or modify state. We always have to return a new state. But there are benefits to this approach –
  + Pure functions are really easy to test.
  + It is really easy to implement features like undo and redo because we always keep the previous state instead of modifying it.
  + It gives us a powerful tool which we call **Time Travel debugging**. So using the tool we can travel back in time and look at our application state as different actions are triggered in the application. We can see how the application state is modified in every step and this makes it really easy to find bugs and fix them.

# Installing Redux to Angular

* There are many implementations for redux but the two common implementations are ngrx/store and ng2-redux (@angular-redux/store). These are the most popular ones and they are very very similar in terms of their API.
* ng2-redux (@angular-redux/store) is built on top of the real redux library and it's compatible with much of the redux ecosystem. It adds bindings for angular too. So you can easily connect angular components with redux.
* ngrx/store on the other hand has gone the route of re-implementing the redux pattern in an Angular 2 and RxJS friendly way. This means it's not compatible with other libraries built for redux.

## Installing ng2-redux / @angular-redux/store

* Create new Angular project
* To install ng2-redux –
  + Angular 2+
    - npm install redux ng2-redux --save
  + Angular 6+
    - npm install redux @angular-redux/store redux-devtools-extension --save
    - @angular-redux/store package needs specific version of redux package. So in case you see any errors on ‘ng serve’, please check compatible version of redux. E.g. @angular-redux/store^10.0.0 worked with redux@4.0.1
  + Here redux is main JavaScript library for redux.
  + And ng2-redux provides an Angular module for dependency injection as well as some other helpers that we're going to use in this section.
* Configuring @angular-redux into our Angular application in order to use it –
  + Create a store.ts file to write our App Reducer function and State interface.
  + E.g. store.ts

/\*\*

 \* this interface will determine what properties we're going to have in our store.

 \*/

export interface IAppState {}

/\*\*

 \* we start with one reducer function here and as our application grows,

 \* we can break down this function into smaller more maintainable functions

 \* each focusing on one domain in the application.

 \*/

export function rootReducer(state, action) {

  return state;

}

* + Import required module and Store  
    import { NgReduxModule, NgRedux } from '@angular-redux/store';
  + Add the NgReduxModule into AppModule’s imports array.
  + Inject NgRedux into the AppModule’s constructor and configure the store with root app reducer and initial state (empty JS object)

export class AppModule {

  constructor(ngRedux: NgRedux<IAppState>) {

    ngRedux.configureStore(rootReducer, {});

  }

}

* + NgRedux is the public interface of @angular-redux/store. It wraps the global redux store and adds a few other add on methods.

# Working with Actions / Dispatching an Action

* With Redux we don’t modify data directly, instead with dispatch an action, this action goes in the store. The store knows our root reducer so it passes the action to the reducer and then the reducer looks at the action and based on that type of action, it will return in new state and then the store will update its state internally.
* E.g.

this.ngRedux.dispatch({ type: 'INCREMENT' });

# The Select Pattern

* To read entire store, we could subscribe to the NgRedux and get State, but this approach we generally don’t use as here we would need manage the subscription and also the code looks tedious.
* E.g.

this.ngRedux.subscribe(() => {

      let state = ngRedux.getState(); // get entire Application State

      this.counter = state.counter; // access required data/slice from the state

 });

* The better approach is to use select decorator.
* To read a state from the store, we need to use select decorator from @angular-redux/store.

E.g. @select() counter;

* By convention, the select decorator gets the name of this field and looks up for a field with the exact same name in the store. If we plan to use a different variable name in the component, we have to pass exact state variable name in the select decorator. E.g. @select('counter') count: Observable<IAppState>;
* @select decorator returns an Observable which we can direct use in our template (.html) using async pipe, so that angular will take care of the subscription.

## Accessing complex State object

* Let’s say we have quite a complex Store object. E.g.

export interface IAppState {

  counter: number;

  messaging: {

    newMessages: number;

  };

}

* And we want to read/access, newMessages property.
* So there are 2 ways, we could access the newMessages property.
  + First approach using array in the select decorator

E.g.

@select(['messaging', 'newMessages']) newMessages;

* + Another approach is using arrow function in the select decorator.

E.g.

@select((s: IAppState) => s.messaging.newMessages) newMessages2;

* + As you can see this is much better as it gives intellisense support to navigate to the desired field.

# Avoiding Object Mutations

* Since our reducer should return new State and not modify existing state, we need to pass copy of the State object, otherwise we would have to set each property in the state in the new returned object (i.e. typing each property present in the state), which is tedious (if we have many properties).
  + We can do it elegantly. There are 2 ways to make a copy of the state object and apply mutations on the fly.
    - Way 1:
      * In javascript, if you want to take a copy of an object we can use Object.assign().
      * We can combine multiple objects into one object with this.

Return Object.assign({}, state, { counter: state.counter + 1 })

* + - * Here the first arg is blank target object which is used to copy all key value pairs, next 2 args are source objects. If the property names are same in the source objects, then the last set value will be used.
      * Problem with this approach is – we can add a property in the 3rs arg above which does not exist in the ‘state’.
    - Way 2:
      * We can fix this by using a typesafe version of Object.assign by installing third party package called tassign.
      * npm install tassign --save
      * After installing this package, we can use it like this –

Import {tassign} from ‘tassign’;

return tassign(state, { counter: state.counter + 1 });

* + - * And if you try to add any new property in the 2nd arg which doesn’t exist in the 1st arg, then we can a compile time error.
      * tassign() works the same way as Object.assing() but with type safety. It doesn’t mutate the arguments, but creates a copy of the object and updates that new object based on values from the arguments.
* Even if we use above ways to avoid mutations, this doesn’t stop us from actually mutating the state object somewhere inside the reducer function. To do so we need to use Immutable objects.

# Immutable Objects

* To prevent accidental mutations, we need to use immutable objects.
* To achieve this, we need to install a third party package called ‘immutable’.
* npm install immutable --save
* After installing, import this in app.module.ts

import { fromJS, Map } from 'immutable';

* And use fromJS function for the initial state and also change Ngedux type to Map<string, any>. This Map is from immutable library. –

export class AppModule {

  constructor(ngRedux: NgRedux<Map<string, any>>) {

    // configuring the store with root app reducer and initial state (empty JS object)

    // fromJS() function gets a plain javascript object and returns an immutable object which is of type ap.

    ngRedux.configureStore(rootReducer, fromJS(INITIAL\_STATE));

  }

}

* fromJS() function gets a plain javascript object and returns an immutable object which is of type Map.
* We also have a map function which does the same thing. However if our object has properties that are complex types, fromJS() will apply immutability on those sub properties as well.
* Next, we now need to update our reducer to make similar changes (changing from IAppState to Map<string, any>).

export function rootReducer(state: Map<string, any>, action): Map<string, any> {

  switch (action.type) {

    case INCREMENT:

      return state.set('counter', state.get('counter') + 1);

  }

  return state;

}

* Also we need to update our component to read/select state –

constructor(private ngRedux: NgRedux<Map<string, any>>) {}

@select((s) => s.get('counter')) counter;

* As you can see, we need to use state.set() to set values in the map. So basically we could add a property which doesn’t belong to the state. (This is the downside of using immutable object)
* So with immutable objects, we prevent accidental mutations but there are some other **drawbacks** like
  + We could add properties which don’t belong to the state.
  + Our IAppState interface becomes useless so we also don’t get intellisense.
  + Since we need to use Map’s get and set methods, chances of making typing mistakes.
* We need to choose between tassign and immutablejs. Both have their own strengths and weaknesses.
* For most of the cases, tassign should be sufficient.

# Redux DevTools

* Redux DevTools gives you a debugging experience that you have never seen before.
* It is just an extension for Chrome, Firefox. Just add it to your browser.
* Once you install this browser extension, we need to do few changes in the app.module.ts file.

## Code Setup to enable Redux DevTools

* Import DevToolsExtension into your app.module.ts.

import { DevToolsExtension } from '@angular-redux/store';

* Now update AppModule constructor

export class AppModule {

  constructor(ngRedux: NgRedux<Map<string, any>>, devTools: DevToolsExtension) {

    // configuring the store with root app reducer and initial state (empty JS object)

    // fromJS() function gets a plain javascript object and returns an immutable object which is of type ap.

    // 3rd arg is to add middleware.

    // middleware is an extension point.

    // So if we can execute some code from the moment an action is dispatched

    // to the moment it reaches a reducer.

    // E.g. logging. we can log every action and do something about it.

    // 4th arg is an array of enhancers, this is where we can use tools extension.

    // this can cause performance issues. So make sure you use only in Devlopment mode.

    var enhancers = isDevMode() ? [devTools.enhancer()] : [];

    ngRedux.configureStore(rootReducer, fromJS(INITIAL\_STATE), [], enhancers);

  }

}

* The 3rd arg to the configureStore() function is to add middleware array. A middleware is an extension point. So if we can execute some code from the moment an action is dispatched to the moment it reaches a reducer.
* The 4th arg is an array of enhancers, this is where we can use tools extension.

This can cause performance issues. So make sure you use only in Development mode.

* Import isDevMode from @angular/core and use this function to configure your store.

import { isDevMode } from '@angular/core';

var enhancers = isDevMode() ? [devTools.enhancer()] : [];

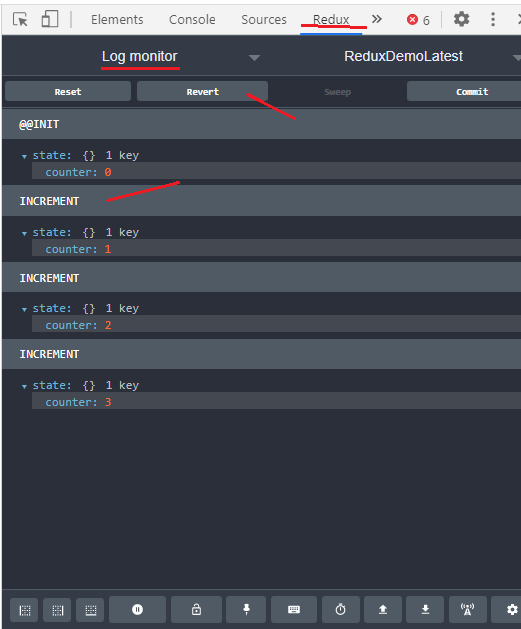
* So here we use enhancers only in development mode.
* Now we are done with the setup.

## Using Redux DevTools in the browser

* Now to do the time travel debugging, go to your browser where your app is running, and open Redux DevTools extension (You can see a tab with the name ‘Redux’ on the same level as ‘Console’, ‘Network’, etc.).
* Once you open ‘Redux’ DevTool, you will see a dropdown with few monitors.

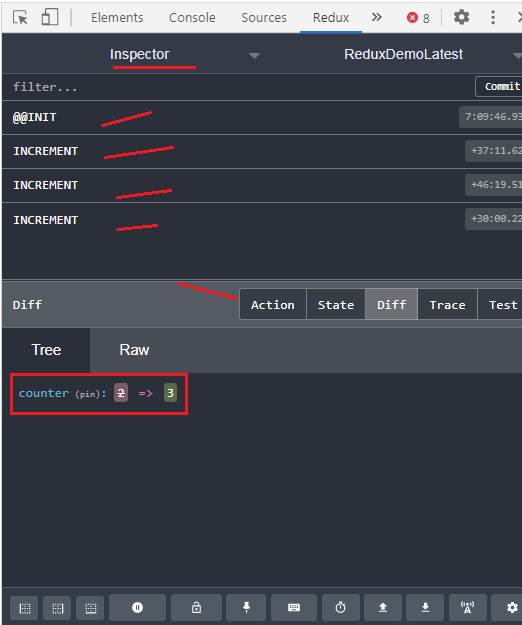
### Log Monitor

* Select ‘Log Monitor’ to see details of each action on the UI. You can revert few actions as well here. You can see current state of the application at each action.



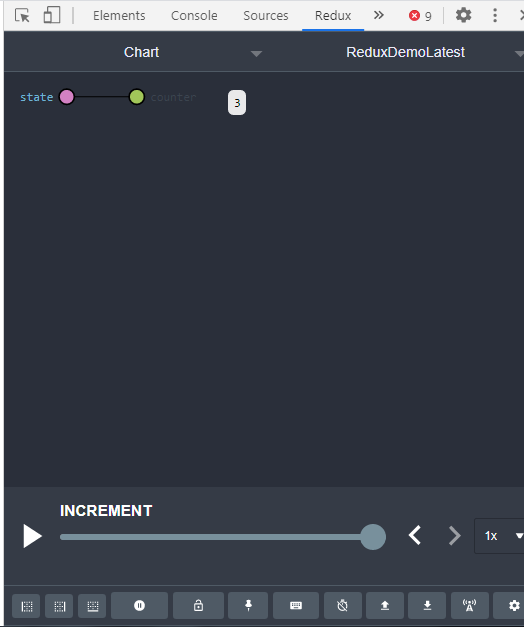
### Inspector

* Next there is ‘Inspector’ monitor,
* It gives you a less detailed view of what happened. You can also see the list of actions that have been dispatched.
* Here we can ‘JUMP’ to any action or ‘SKIP’ any action to see State behavior.
* Once we select any action, then on the next tab you see its details like Action, State (after the action), Diff (before and after), etc.



### Chart monitor

* Next you can see ‘Chart’ monitor,
* With this, we can also see a visual representation of application state and how it changes.
* We can do time travel using the slider and can see state at that point of time.
* Extremely powerful.
* Also there is a button to store (in JSON) and then reload that application state. This is extremely powerful because you can come back later and reload the saved application state.
* So with this, you can bring the application into a state where you have a bug and save the state and then pass that state file to your colleague developer and they can jump start debugging instead of working with a complex form adding lots of data every single time they load the page.
* There is also a button to persist the current state of the application. With that if you refresh your page, you will be on that persisted state directly.



* So this tool gives you a very detailed look into how the application state is changing from each action to another.
* Without this tool, if you want to get this visibility, we have to throw lots of console.log statements all over the code.

# Calling Backend APIs

* In typical non-redux application, we make backend HTTP service calls and subscribe to the HTTP result in the component and store the result of the service call in the component variable.
* In Redux, we don’t store the state in the component, instead on subscribe, we dispatch an action and then that action goes to a reducer and then eventually the state of the store gets updated under the hood.
* In doing so, we should also use redux to maintain showing loading icon.
* For a create operation, if we are first generating ID at the client side and then doing backend HTTP call, it is called optimistic update. In other way around if we are first doing backend HTTP call where ID is generated e.g. in DB, and then updating the client state, this is called as pessimistic update. Pessimistic update is better in my opinion as this is full proof in case of any errors during the HTTP calls.
* Redux is for maintaining the **client state** and conceptually it should not worry about the server state.
* The concept of action creators as a solution to talk to the server is an anti-pattern and it brings extra complexity in your implementation. So stay away from it and use the approach I showed you in this lecture.

# Refactoring Fat case statement in the Reducer

* As we add more and more actions, as a result our switch-case in reducer becomes fat (too many lines of code) and we can’t see details of one case block on single window (we have to scroll most of the times).
* There is a simple technique to refactor this code and make it more maintainable.
* So basically we're going to use extract function refactoring.
* So basically, we should extract the entire case block into an internal function which will take 2 arguments – state and action   
  e.g. function addTodo(state, action). And then in the main reducer function just call ‘return addTodo(state, action);’ for the corresponding case statement.
* With this simple change, our main reducer will look short and sweet.
* Now if we can see, the argument state and action are repeating for each functions in the all the case statements. We can take this to the next level.
* Potentially we could put all these functions in a class (without any arguments) and pass the state and action in the constructor of that class. This way you don't have to pass state and action parameters in every line in the case statement.

# Dealing with Complex Domains

* As your application grows and you introduce new actions you'll end up with a fat reducer and you will end up with a fat Switch block with so many cases. We don't want to have 50 or 100 cases here.
* So when your application grows to that size that's most definitely an indication that you need to break down your application into smaller and more maintainable parts or modules.
* So in every module we want to have an **interface** that determines the state of that module, we want to have an **initial state** object and a **reducer** that is specifically designed for that module.
* Also actions specific to each reducer should be in separate actions.ts file.
* And in the end, the state interface and initial state should be added in the main app/root reducer file’s state interface and initial state. And remove rootReducer function, we will have to combine all sub reducers into root reducer.
* And now we need to combine all the reducers in the root reducer. For that, import combineReducers function from redux library and write this –

import { combineReducers } from "redux";

export const rootReducer = combineReducers({

  tasking: taskingReducer,

  messaging: messagingReducer

});

* When we have multiple reducers, make sure you initialize the state argument to initial state, otherwise we would get runtime error. This is a requirement from Redux library. E.g.   
  export function taskingReducer(state: IAppState = INITIAL\_STATE, action): IAppState {…}
* It is always better to do this for all your reducers even if you have only one reducer.
* Now in the components where we are getting state from reducers, we beed little modifications. E.g.

@select((s) => s.tasking.todos) todos;