Advanced Angular & Rxjs

Kobi Hari & Shimon Dahan

hari@applicolors.com
shimond@any-techs.co.il

Reactive X

Rx - Motivation

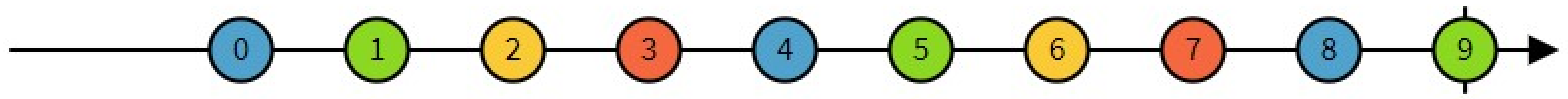
- Mostly useful when using single point of data with multiple consumers
- The traditional solution to this is using method invocation to pull data
- There are several problems with this approach:
  - Redundancy - Multiple method invocation to “calculate” the same data
  - Synchronization – When should each consumer ask for data
- Reactive X moves the initiative to the producer. Instead of being asked for data, it pushes it to the consumer.
- The producer Acts, the consumer Re-acts.
- **Hence Reactive…**

Some Buzzwords

- Observer Pattern
- Iterator Pattern
- Asynchronous Development

Streams

A Stream is a sequence of events describing the state of a single data over time.



- Streams may end successfully
- Streams may end in failure
- Streams may never end…

Observers

An `Observer<T>` is an object that get's notified regarding a stream of `<T>`. Notifications are done using the following methods

`Next(value: T)`	called to notify the observer of the new value of the data
`Complete()`	called to indicate that the stream is over – so the value will no longer change
`Error(err: any)`	called to indicate that there is something wrong with the stream – and that there will be no more events, but for a bad reason

- Of course, `Observer<T>` is an **interface** to be implemented so your object may be called about a stream.

Observable<T>

An `Observable<T>` is an object that allows multiple Observers<T> to subscribe.

`Subscribe(observer: Observer<T>)`	Subscribe another observer

Warm and Cold Observables

- An Observable is lazy
- It does not really start unless you subscribe.
- In fact – the only logic you can provide into an observable is on the subscribe callback

A “Cold” Observable is an observable that hasn’t been subscribed to yet.

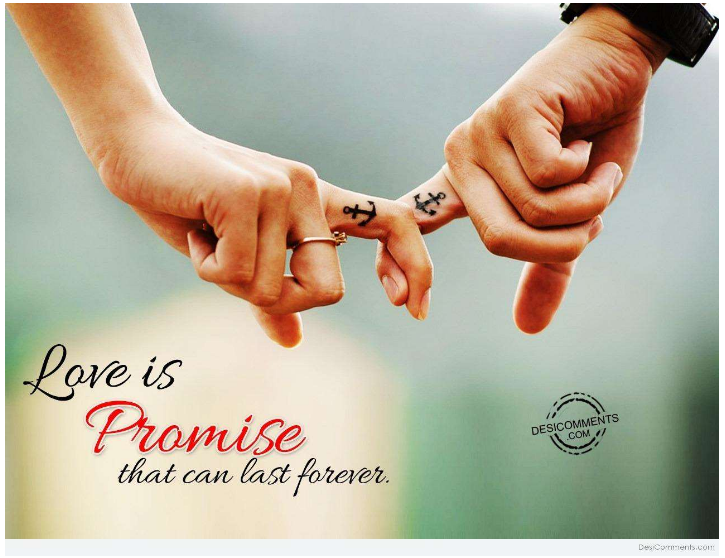
DEMO

Demo Conclusions:

- You can not run any logic in an observable, except that one that decides how to treat new observers when they subscribe.
- This is the **only way to access observers** and raise events.
- Observables will only execute when they are subscribed to.
- And they will re-execute every time they are subscribed again.

Is it like promises?

- Well yes and no.
- A promise is like an observable that only throws one next, and always throws complete right after the single next.
- In a way – ~~love~~ (rxjs) is a promise that lasts forever 😊
- Oh and promises are not lazy… they run code as soon as the promise is created even if it is never listened to. (“then”-ed to)



Observables may send different events to each observer

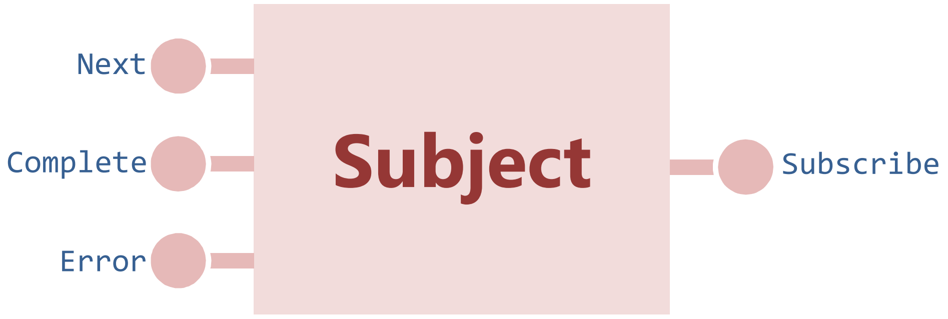
Accessing an observer is possible only in the constructor

A constructed observer is "read only". You can subscribe to it, but you can not emit events

Subjects

A Subject is an observable that allow you to emit events.

It is both an Observable and an Observer at the same time



- When you call “next”, “complete” or “error”, it sends the event to all subscribed observers
- In a way, it is like a writeable observable

Behavior Subject

- `BehaviorSubject<T>` **`extends`** `Subject<T>`
- When subscribed to, it emits the latest event to the subscriber
- Ideal for storing variable state
- Requires an initial value on constructor (why?)
- Has a `Value` property you can use to get the value of the latest event.

DEMO

Unsubscribe !

- Consider a specific scenario
  - Single service
  - Component subscribes to service
  - Component gets destroyed and re-created many times
- Service observable now points to component (in the callback)
- It accumulates callbacks
- Memory leaks
- Probably even more problematic: performance leaks
- Possibly also – problematic responses to changes
- Some services do not require unsubscribe:
  - `http`
  - `activatedRoute`

Async Pipe

- A new method of development where you bind directly to the observables in the component.
- It Subscribes and Unsubscribes for you – which avoids memory leaks
- Also works with promises

Redux

with

`@ngrx/store`

Redux + RxJS + Angular = NgRX

Our Goal

- Understand Redux
- Understand how to combine Redux with Angular
- Understand how combining the two can improve performance

Our Objective

- Following our Redux lesson, our objective:
    - Create a todo list app
    - Entire app with chage detection strategy OnPush

What is a State

- In college at automation course we learned about Finite-state machine (FSA)
- FSA is a mathematical model of computation
- FSA can be in one state out of a group of final states at any given time
- FSA can transition to a new state based on certain change in input

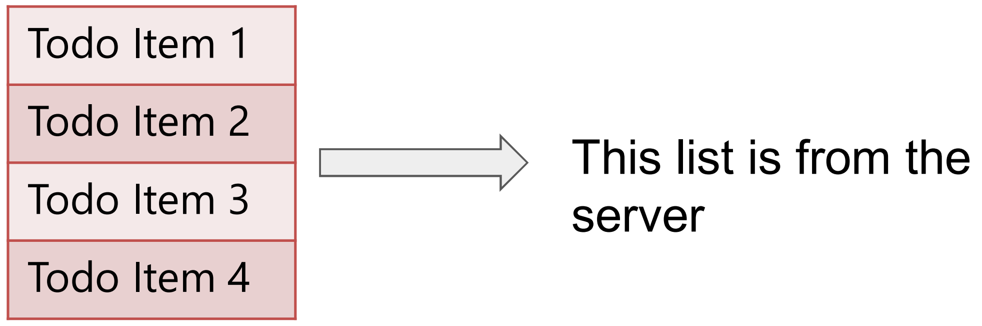
$$\begin{tikzcd}
{} \arrow[r] & S1 \arrow[rr, bend left, "a"] & & S2 \arrow[ll, bend left, "b"] \arrow[loop above, "a"] \arrow[dr, "c"] & \\
 & & S3 \arrow[ul, "a"] \arrow[rr, bend left, "b"] & & S4 \arrow[ll, bend left, "d"]
\end{tikzcd}$$

Can we look at our frontend application as an FSA?

What can be the cause of a transition in state in a frontend web app?

Todo App State

- In our todo app we have a component that displays the list of todo items
- At the beginning, the list is empty and we query the server for the list of items
- After the server return our response we display the list of tasks
- The todo list component looks like this:



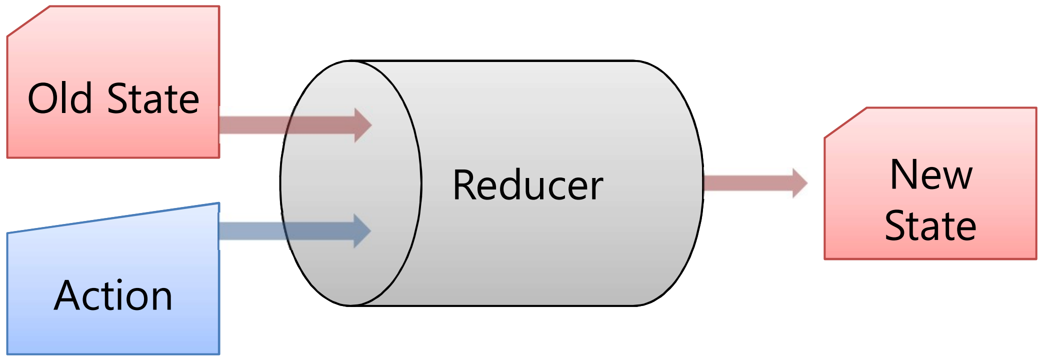
What is Redux

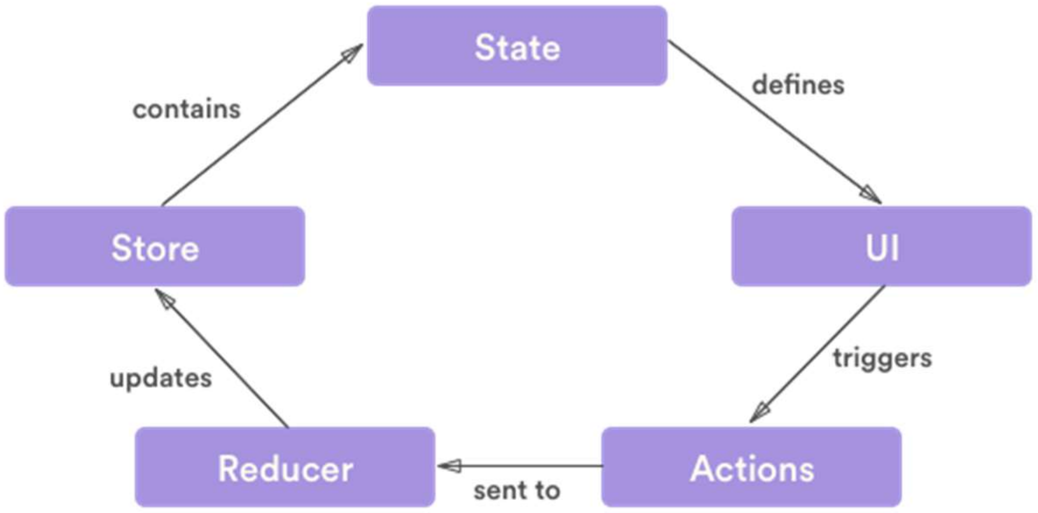
- Predictable state container for JS apps
- Using redux we have a single object holding the state of our app which is called **store**
- The state in the store can be an Object, Array, Primitive but it’s highly recommended that it will be **immutable**
- The only way to change the state is by calling a method on the store called **dispatch**
    - **store.dispatch(action)**
- An action is a simple object describing **what happened**
- The store decides how the state will change using a pure function called reducer
- The reducer answers the questions **How does the state change?**
    - **(state, action) => state**

Redux

Redux core concepts are:

- Actions
- Reducers
- Store




State
defines
UI
triggers
Actions
sent to
Reducer
updates
Store
contains

Actions and Action creators

- **Action** is a simple JS Object
- The action needs to hold the following information
  - identifier of the action (string)
  - data associated with the action
  - example: `{type: ‘SET_LOADING’, payload: true }`
- Action creator is a pure function which returns an action

```
function setIsLoading(isLoading) {
    return {type: ‘SET_LOADING’, payload: isLoading};
}
```

Which action creators do you think our Todo app requires?

Reducers

- Reducers are functions that apply action to create a new state
- Reducers are **pure** functions
- Reducers get the current state and action as arguments
- The reducer need to decide based on the action and the current state what the next state will be
- The state has to be **immutable.**
- When our state grows if we use only a single reducer the reducer will be huge so it is better to split the state to sections and also split the reducers
- Redux has a function called **combineReducers** to help us split the app and make each reducer in charge of certain section of the state

combineReducers

- When our state grows so will our single reducer
- We want to split the state to logical sections
- Let’s say our todo application will need to hold information about the current user
- We will create another reducer that will handle the user info and combine those reducers with the `combineReducers` function

Store

- there is a single store in redux application
- the store holds the state of our app
- to create the store we use `createStore`
  - `createStore(reducers, initialState)`
- we need to pass to the `createStore` the combined reducers and optional initial state
- Let’s create our store

Redux dev tools

- one of the strong features of redux is the ease of testing and ease of development
- you have a state and the app should behave according to the state
- for development purpose it’s better if we can easily examine the current state and all the actions that got us to this state
- there is an easy to use browser extension:
    - https://github.com/zalmoxisus/redux-devtools-extension
- to install the devtool extension:

```
npm install redux-devtools-extension --save-dev
```

Combining with Angular

- Now that we know what Redux is, lets try and combine Angular and redux together using **@ngrx/store**

What is @ngrx/store

- state management for Angular applications inspired by redux
- we select items from the state and get them as observables
- we then use async pipe to display them in the components
- async pipe will cause a re-render even if the component is in change strategy on push
- install with npm:

```
npm install @ngrx/store --save
```

Store as a Module

- Similar to how we placed routing in separate modules, we will do the same with our state
- Each module which needs to add items to our state will have that logic in a store module
  - **app-store** - will contain store module for the root module
  - **user-store -** will contain store module for the feature module called user
- Inside each module we can split the reducers based on logical sections

Actions

- Let’s start by implementing our actions
- We will hold string types of the actions in a class with static properties called `TodoActionsTypes`
- Each action will implement Action interface
- We will also use type alias to name the type of actions

Reducer

- We will define an interface that describes the section of the state that the reducer is in charge of
- We will define our initial state
- Create our reducer as a pure function with switch and case
- The pure function will use the types defined in the action file

Combining Reducers

- We will create a file called **reducers**
- In this file we will describe the entire state of the app-store module
- We will create `ActionReduceMap<AppInterface>` which is a dictionary which maps keys to reducers

Store Module

- We will call **StoreModule.forRoot** to create our store
- It will get the reducer map we created before
- For lazy loaded module their state will be lazy loaded as well and we will call the function **StoreModule.forFeature**
- We will add our store module to the app module
- We can now inject the store to our services and components

Modifying the Service

- We can inject the store into our Todo service the store
- In the **map** operator when we get the tasks from the server we can call the **dispatch** method with our action to add the tasks to the state
- In the app component call the service get tasks to initiate the state with the tasks from the server

Todo List Component

- Create a component to display the list of todo tasks
- Inject the store to the component
- We will use the select method on the store to select certain properties from the state
- The property will return to us as observable
- We will use the async pipe to display that observable

Summary

- With redux we can manage the state of our app
- The only way to change the state is by calling **store.dispatch(action)**
- Reducers will decide how the state changes
- Combining Angular and redux is a powerful tool which gives us
  - More testable app
  - Better performance
  - Easier management of the data of our app

DevGeekWeek 2019
INSPIRED BY INNOVATION

Thanks :)

See you next year