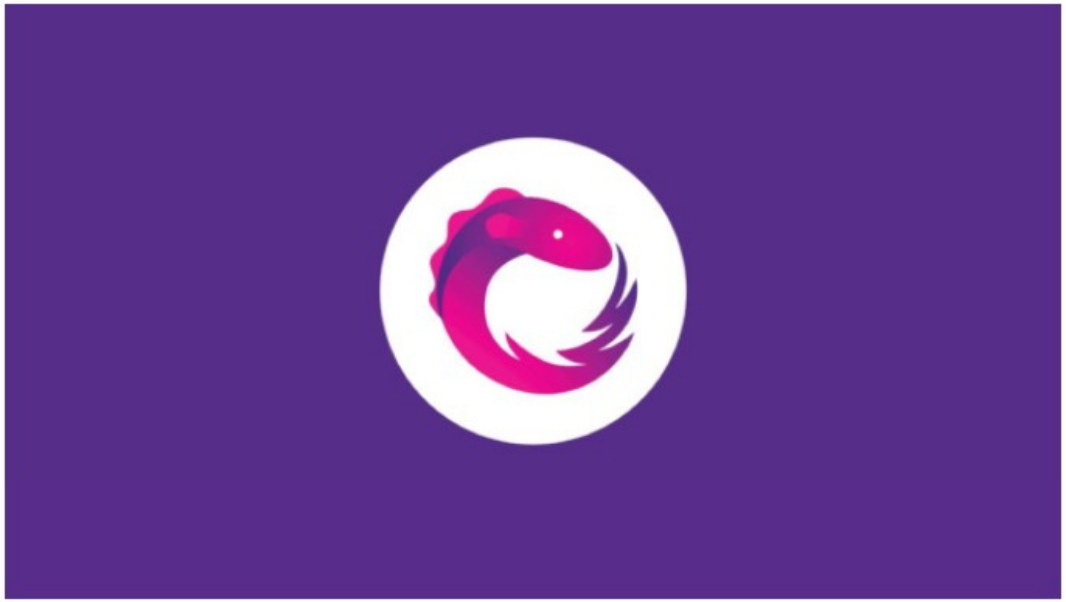
Rx.JS — Things that you should understand before start.

Banujan Balendrakumar Follow

Sep 5 · 5 min read



If you have ever worked with **Angular**, You probably might have seen something like **Observables**, **Subscribe()**, **Subject** and a few more. Sometimes we may be using it without knowing what it is. If you haven't familiar with those terms. Don't worry, This is **not** something about **Angular**. This is all about an **awesome** and **powerful** Javascript library that you should learn.

Let's jump into the topic 🚀

Observer Pattern 🧐

Observer Pattern is one of the **Software Design Patterns**. If you don't have any idea about this **Observer Pattern**, I highly recommend you to have a look and come back for a better understanding. Because I am not going to drag this story by explaining it. Anyway, Let me give a quick introduction.

Observer Pattern is a Design Pattern that enforces **Event-Driven Application Development**. This pattern mainly has two objects, One is "**Subject**" and another one is "**Observer**". Basically **Subject** maintains a list of **Observers**. It will notify them when the state changes.

> ***High-Level Example***
>
> *The **YouTube** (**Application**), Has more **channels** (**Subjects**).*
> ***We** (**Observers**) subscribe to one of our favourite **channels** (**Subject**).*
> *So, Whenever the **channel** (**Subject**) publishes new **content** (**Data/State**), It will notify **us** (**Observer**).*

As simple as that. 😉

Reactive X ⚡

> *ReactiveX is a library for composing asynchronous and event-based programs by using observable sequences.*

ReactiveX is a library that allows us to extend above **Observable Pattern**

and build **Reactive**, **Asynchronous** and **Even-Based Applications**. ReactiveX provides wide languages support. **RxJS** is the javascript version of it. You can find all **other language** support here.👇

Rx.JS 🔥

As I mentioned, **RxJS** is the Javascript library that extends **ReactiveX**. Before we getting started to code with **RxJS**, We should clearly understand its key objects. Otherwise, we cannot use it efficiently. **RxJs** is all about these **3O**s and **3S**s,

1. **O**bservable 🔊
2. **O**bserver 🦹
3. **O**perators 🛠
4. **S**ubscription 🔔
5. **S**ubject 🕳
6. **S**cheduler ⏰

1. Observable

Observables are the object that **push/stream** the data to their **Observer**.

> *We can **eat** only **eatable** objects. Also, we can **move** only **movable** objects. Like*

> *that, We can only **Observe**, The **Observable** objects.*

So in **Rx.JS**, If we want to **Observe** an object's changes, The object should be **Observable.** In order to execute the **Observable**, It needs at least one **Subscriber**.

Observables are **unicast**. That means it can only send data to **One Observer**. If you subscribe **Multiple Observers** to **Observable**, It will be invoked multiple times for each **Observer**.

Observables are **uni-directional**. That means It can only pass data from **Observable** to **Observer**. We cannot feed data into **Observer** from the outside.

```
import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {

  subscriber.next("First State / Data of Observable"); // Notifiying first state
  subscriber.next("Second State / Data of Observable"); // Notifiying second state
  subscriber.next("Third State / Data of Observable"); // Notifiying third state

  // Just waiting 5 seconds

  setTimeout(() => {
    subscriber.next("Fourth State / Data of Observable"); // Notifiying fourth state
    subscriber.complete(); // Notifiying that the is no more data to notify
  }, 5000);

});
```

medium-rxjs-observable.js hosted with ❤ by GitHub view raw

2. Observer

An **Observer** is an object that subscribes to **Observables** so it can be called

by **Observables** to emit the data.

> *We subscribe to **Newsletters**, So we can get updates. In this case, We are **Observers** and Newsletter is **Observable**.*

An **Observer** object will have 3 methods **next()**, **error()** and **complete()**. These methods will be called by the **Observable** that it subscribes.

next(param)

This method will be called by the **Observable** to send new data. It accepts one **parameter** as the **data** sent by the **Observable**. The data can be anything such as ***Number***, ***String***, ***Array***, ***Object*** & etc.

error(param)

Same as **next()**, But here **Observable** calls this method in order to notify any **error**. This method also accepts one **parameter** that is an **error** message from the **Observable**.

complete()

When the **Observable** is done with notifying and if there are no events left, It will call this method to notify that there is nothing to emit. This method doesn't accept any parameters.

```
1  // Simple Observer Object
2
3  const observer = {
4    next(data) {
5      console.log('We Received new data ' + data);
6    },
```

```
  error(err) {
    console.error('Oh, No. We got an error : ' + err);
  },
  complete() {
    console.log('Done. Nothing more.');
  }
};
```

medium-rxjs-observer.js hosted with ❤ by GitHub view raw

3. Operators

Let's talk about this at the end of this story. 😊

4. Subscription

An **Observer** or **Observable** cannot do anything by itself. In order to work with the **Observable**, An **Observer** should subscribe to it. A **subscription** is an object that holds the execution of an **Observable**. This subscriber has one method called **unsubscribe()**. This method just cancels the existing subscription. So the **Observable** will not execute the **Observer** anymore.

Also, A **subscription** may have another **child subscription**. In that case, If you **unsubscribe** the parent one, Then the child will be unsubscribed too.

```

/**
First Example
**/

import { Observable } from 'rxjs';

// Observable Object
const observable = new Observable(subscriber => {
  subscriber.next("First Message");
  subscriber.next("Second Message");
});

// Observer Object
```

```
const observer = {
  next: (data) => console.log('We Received new data ' + data),
  error: (err) => console.error('Oh, No. We got an error : ' + err),
  complete: () => console.log('Done. Nothing more.')
};

// Subscribing
const subscription = observable.subscribe(observer);

// Unsubscribing
subscription.unsubscribe();
```

medium-rxjs-subscription-1.js hosted with ❤ by GitHub view raw

```

/**
Second Example
**/

import { Observable } from 'rxjs';

// First Observable Object
const observable = new Observable(subscriber => {
  subscriber.next("First Message");
});

// Second Observable Object
const observable2 = new Observable(subscriber => {
  subscriber.next("Second Message");
});

// first subscription
const subscription = observable.subscribe(data => console.log("We got a data " + data));

// second subscription
const subscriptionChild = observable2.subscribe(data => console.log("We got a child data " + dat

// adding 2nd subscription as child of 1st
subscription.add(subscriptionChild);

// unsubscribing parent BUT child also will be unsubscribed
subscription.unsubscribe();
```

medium-rxjs-subscription-2.js hosted with ❤ by GitHub view raw

5. Subject

Subjects are a special type of **Observable**. It can act as both **Observable** and **Observer**. That means you can use a **Subject** as **Observable** as well as **Observer**.

Subject as an Observable

Unlike the normal **Observable**, **Subjects** are **Multicast**. It can stream the data to multiple **Observers** without maintaining separate executions.

Subjects as an Observer

Subjects are **bi-directional**. You can pass data from **Subject** to **Observer** as well as **Observer** to **Subject**. You can simply feed data into the Subject by calling the **next()** method.

```
import { Subject } from 'rxjs';

const subject = new Subject();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});
subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next("First Data");
subject.next("Second Data");

// CONSOLE OUTPUTS ---------------------

// > observerA: First Data
// > observerB: First Data
// > observerA: Second Data
// > observerB: Second Data
```

medium-rxjs-subjects.js hosted with ❤ by GitHub view raw

Subject vs Observable

```
1  import { Subject, Observable } from 'rxjs';
2
3  const observable = new Observable(subscriber => {
4     subscriber.next(Math.random()); // passing a random number
5  })
6
7  // Gives 2 different number. Because Observables are unicast and it will be execute separatly fo
8
9  observable.subscribe(data => {
10   console.log('subscription a :', data); //subscription a :0.2859800202682865
11 });
12
13 observable.subscribe(data => {
14   console.log('subscription b :', data); //subscription b :0.694302021731573
15 });
16
17
18
19 const subject = new Subject();
20
21 // Getting same data for both subscription, Because Subjects will be executed once and
22 // multicasting the data to multiple observers.
23
24 subject.subscribe(data => {
25   console.log('subscription a :', data); // subscription a : 0.91767565496093
26 });
27
28 subject.subscribe(data => {
29   console.log('subscription b :', data); // subscription b : 0.91767565496093
30 });
31
32 // passing from outsite. Yes we can do it for subject becuase it is bi-directional
33 subject.next(Math.random());
```

medium-rxjs-subject-vs-observable.js hosted with ❤ by GitHub view raw

Operators

> *Let's discuss the skipped part Operators.*

Operators are very important and helpful. These are just helper functions that make the complex async to be easily composed.

There are 2 types of **Operators**.

1. **Pipeable Operators**
2. **Creation Operators**

Pipeable Operators can be attached to an **Observable** using **pipe()** method. It will not change the existing **Observable Object**. Instead, It takes the existing **Observable** and outputs a **new Observable**.

```
import { Observable } from 'rxjs';
import { first } from 'rxjs/operators';

// Create a Observable which emits 2 values
const observable = new Observable(subscriber => {
  subscriber.next('First Message');
  subscriber.next('Second Message');
});

// "first()" is a pipeable operator that only returns the observable with the first value.
observable
  .pipe(first())
  .subscribe(data => console.log(data));

// CONSOLE OUTPUT ------------------
// > First Message
```

Creation Operators are standalone functions that can create **Observables** from standard predefined behaviours or by combining other **Observables**. These operators **cannot** be **piped** with **Observables**.

```
import { of } from 'rxjs';

// of is a Creation Operator,
// That can create new Observable by taking the parameters as the emitting values.

of("First Data", "Second Data")
  .subscribe(data => console.log(data));

// CONSOLE OUTPUT ------------------------
// > First Data
// > Second Data
```

medium-rxjs-operators-creation.js hosted with ❤ by GitHub view raw

Apart from **Of()** and **First(),** there are a lot of **Operators** that I can't explain in this story. But I am pretty sure that If you understand this basics, Then you can easily understand others as well. Find more useful **Operators** below.

RxJS Operators: https://rxjs.dev/guide/operators

5. Scheduler

The **scheduler** is a control mechanism that takes care of, when a **subscription** starts and when the **subscriber** gets notified.

A **Scheduler** consists of the following components:

1. **Data Structure:** The scheduler knows how to store, arrange and queue the tasks based on their priority or any other ruleset.

2. **Execution Context**: The scheduler can control whether the task should be executed immediately or should be passed into another mechanism as a callback.
3. **Virtual Clock**: The tasks beings scheduled on a scheduler will follow the time that is denoted by the clock.

If you don't understand the **Scheduler** concept. Don't worry, It is not that necessary to start **Rx.JS**. I am stopping here because I want to make this story as basic and clear as possible. Just have look at this code so you will have a basic idea of how the scheduler works.

```
import { Observable, asyncScheduler } from 'rxjs';
import { observeOn } from 'rxjs/operators';

// "asyncScheduler" is a type of Scheduler which executes the Observable Asynchronously.
// "observeOn" is a Pipeable Operator that help us to re-emit the data with Scheduler.

const observable = new Observable((observer) => {
  observer.next("First Data");
  observer.next("Second Data");
  observer.next("Third Data");
}).pipe(
  observeOn(asyncScheduler)
);

console.log('just before subscribe');

// Since we are using "asyncScheduler", This code will be executed asynchronously.
observable.subscribe({
  next(data) {
    console.log(data)
  }
});

console.log('just after subscribe');

// CONSOLE OUTPUT
// > just before subscribe
// > just after subscribe
```

```
29  // > First Data
30  // > Second Data
31  // > Third Data
```

medium-rxjs-scheduler.js hosted with ❤ by GitHub view raw

RxJS Scheduler: https://rxjs.dev/guide/scheduler

Conclusion

Rx.JS is the best library to build **Asynchronous** and **Event-Driven** Applications on Javascript. It is a standalone library so you can directly use it with any javascript framework or library. **Rx.JS** comes with Angular by default. In Angular, **Rx.JS** is very helpful to stream notifications and states across components.

I tried my best to keep this story **beginner-friendly**. I ignored the most confusing and advanced concepts. It is totally fine, If you understand what we discuss so far then you can start working on **Rx.JS** also you are capable of understanding advanced topics. Hope you find this useful.

Give few claps 👏, If you really like this.

Happy Coding… ❤

Rxjs Reactivex Observables Event Driven Programming JavaScript

WRITTEN BY

Banujan Balendrakumar

Software Engineer | AWS Certified Solution Architect | Auth0 Ambassador | Blogger | Stackoverflower

Follow

SLIIT FOSS Community

SLIIT FOSS Community is a team of volunteers who believe in the usage of Free/Open Source Software (FOSS). The primary objective of the community is to promote, develop and diversify the usage of Free/Open Source Software at SLIIT.

Follow

More From Medium

More from SLIIT FOSS Community

Protobuf : What? and Why?

Banujan Balendrakumar in SLIIT...
Nov 8 · 4 min read

50

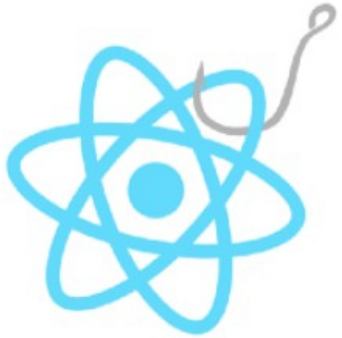
More from SLIIT FOSS Community

Top 6 HTTP Headers that improve your site security.

Banujan Balendrakumar in SLIIT...
Oct 16 · 4 min read

77

More from SLIIT FOSS Community



Forms with React Hooks

Methmi in SLIIT FOSS Community
Oct 14 · 10 min read

2

Learn more.

Medium is an open platform where 170 million readers come

Make Medium yours.

Follow the writers, publications, and topics that matter to you,

Write a story on Medium.

If you have a story to tell, knowledge to share, or a

to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

and you'll see them on your homepage and in your inbox. Explore

perspective to offer — welcome home. It's easy and free to post your thinking on any topic. Start a blog