**RxJS Observable:**

Aug 16, 2022

<https://www.tektutorialshub.com/angular/angular-observable-tutorial-using-rxjs/>

Data Steams:

**Value:** i.e the next value in the stream  
**Complete**: The stream has ended  
**Error**: The error has stopped the stream.

The reactive programming is all about creating the stream, emitting value, error or complete signals, manipulate, transfer or do something useful with the data streams.

The [RxJS](https://rxjs.dev/guide/overview) (Reactive Extensions Library for JavaScript) is a javascript library, that allows us to work with asynchronous data streams

The Angular uses the RxJS library heavily in its framework to implement Reactive Programming. Some of the examples where reactive programming used are

* Reacting to an [HTTP request in Angular](https://www.tektutorialshub.com/angular/angular-httpclient/)
* [Value changes](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/) / [Status Changes](https://www.tektutorialshub.com/angular/statuschanges-in-angular-forms/) in Angular Forms
* The Router and Forms modules use observables to listen for and respond to user-input events.
* You can define custom events that send observable output data from a child to a parent component.
* The HTTP module uses observables to handle AJAX requests and responses.

The RxJs has two main players

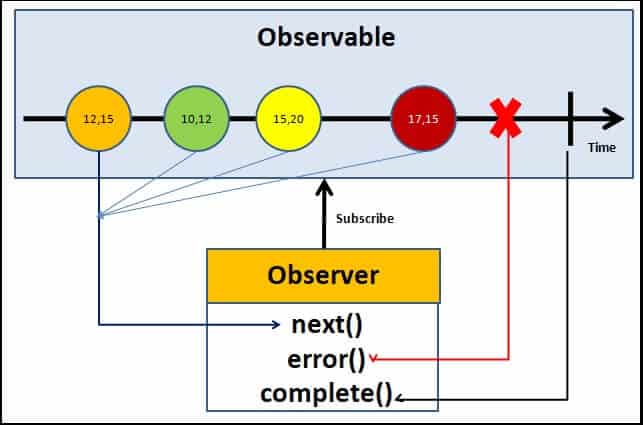
1. Observable
2. Observers ( Subscribers)

Observable is a function that converts the **ordinary stream of data** into an **observable stream of data**. You can think of Observable as a wrapper around the **ordinary stream of data**.

**Observable stream** or simple Observable emits the **value from the stream** asynchronously. It emits the **complete** signals when the stream completes or an **error** signal if the stream errors out.

Observables are declarative. You define an observable function just like any other variable. The observable starts to emit values only when **someone subscribes to it**.

The observer must subscribe with the observable to receive the value from the observable. While subscribing it optionally passes the three callbacks. next(), error() & complete()



* Observers/subscribers subscribe to Observables
* Observer registers three callbacks with the observable at the time of subscribing. i .e next(), error() & complete()
* All three callbacks are optional
* The observer receives the data from the observer via the next() callback
* They also receive the errors and completion events from the Observable via the error() & complete() callbacks

There are many operators available with the RxJS library, which makes the task of creating the observable easy. These operators help us to create observable from an array, string, promise, any iterable, etc. Here are list some of the commonly used operators

* [create](https://www.tektutorialshub.com/angular/rxjs-observable-using-create-of-from-in-angular/)
* defer
* empty
* [from](https://www.tektutorialshub.com/angular/rxjs-observable-using-create-of-from-in-angular/)
* [fromEvent](https://www.tektutorialshub.com/angular/create-observable-from-event-using-fromevent-in-angular/)
* interval
* [of](https://www.tektutorialshub.com/angular/rxjs-observable-using-create-of-from-in-angular/)
* range
* [throwError](https://www.tektutorialshub.com/angular/using-throwerror-in-angular-observable/)
* timer

Subscribing to the observable:

ngOnInit() {

**this**.obs.subscribe(

      val => { console.log(val) }, *//next callback*

      error => { console.log("error") }, *//error callback*

      () => { console.log("Completed") } *//complete callback*

    )

}

Create a new Observable:

obs = **new** Observable((observer) => {

    console.log("Observable starts")

    setTimeout(() => { observer.next("1") }, 1000);

    setTimeout(() => { observer.next("2") }, 2000);

    setTimeout(() => { observer.next("3") }, 3000);

    setTimeout(() => { observer.next("4") }, 4000);

    setTimeout(() => { observer.next("5") }, 5000);

  })

obs = **new** Observable((observer) => {

    console.log("Observable starts")

    setTimeout(() => { observer.next("1") }, 1000);

    setTimeout(() => { observer.next("2") }, 2000);

    setTimeout(() => { observer.next("3") }, 3000);

    setTimeout(() => { observer.error("error emitted") }, 3500);    *//sending error event. observable stops here*

    setTimeout(() => { observer.next("4") }, 4000);          *//this code is never called*

    setTimeout(() => { observer.next("5") }, 5000);

  })

obs = **new** Observable((observer) => {

    console.log("Observable starts")

    setTimeout(() => { observer.next("1") }, 1000);

    setTimeout(() => { observer.next("2") }, 2000);

    setTimeout(() => { observer.next("3") }, 3000);

    setTimeout(() => { observer.complete() }, 3500);   *//sending complete event. observable stops here*

    setTimeout(() => { observer.next("4") }, 4000);    *//this code is never called*

    setTimeout(() => { observer.next("5") }, 5000);

  })

Unsubscribing to close the observable when we no longer require it. If not it may lead to memory leak & Performance degradation.

ngOnDestroy() {

**this**.obs.unsubscribe();

}

**this**.**data**$ = Observable.of(JSON.parse(`{

"data": [

{

"firstname": "abc",

"lastname": "aa"

},

{

"firstname": "xyz",

"lastname": "bb"

}

]

}`).subscribe(res=> console.log(res.**data**));

An Observable is an entity that emits (or publishes) multiple data values (stream of data) over time and asynchronously.

Observable represents the idea of an invokable collection of future values or events.

Observer is a collection of callbacks that knows how to listen to values delivered by the Observable.

[Subscriptions](http://reactivex.io/rxjs/class/es6/Subscription.js~Subscription.html) are objects that are returned when you subscribe to an Observable.

Subscription represents the execution of an Observable, is primarily useful for cancelling the execution.

A [Subject](https://rxjs-dev.firebaseapp.com/guide/subject) is a special type of Observable that observers can also subscribe to it to receive published values but with one difference: **The values are multicasted to many Observers**.

By default an RxJS Observable is unicast.

RxJS provides two other types of Subjects: BehaviorSubject and ReplaySubject

You can create an RxJS Observable using the Observable.create() method which takes a function with an observer argument. You can then subscribe to the returned Observable instance.

After creating an Observable, you can subscribe to it using the subscribe() method on the instance which returns an instance of Subscription.

Example:

1. let ob$ = Observable.create((observer) => {

observer.next("A new value!");

});

2. let observer = {

next: data => console.log( 'Data received: ', data),

complete: data => console.log('Completed'),

};

3. let subscription = ob$.subscribe(observer);

Angular 10 uses the RxJS Observable as a built-in type for many of its APIs such as:

* The HttpClient methods return Observables and actual requests are only sent when you subscribe to the returned Observable.
* The Router uses Observables in multiple places such as:
  + the [events](https://angular.io/api/router/Router#events) of the Router instance is an Observable to listen to events on the router.
  + Also ActivatedRoute (which contains information about the route associated with the currently loaded component on the router outlet) has many Observable properties such as params and paramMap for the route parameters.
* The Reactive Forms Module uses reactive programming and Observables for listening to user input.
* The @output() decorator in a component takes an EventEmitter instance. EventEmitter is a subclass of the RxJS Observable.

After calling the getItems() method on the component we can use the async pipe in the component template to subscribe to the returned Observable

Some notes about the stream:

https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html

<https://www.baeldung.com/java-8-streams>

all code in the repository tutorials-master.zip

**We can only use one terminal operation per stream.**

The correct and most convenient way to use streams is by a **stream pipeline, which is a chain of the stream source, intermediate operations, and a terminal operation:**

List<String> list = Arrays.asList("abc1", "abc2", "abc3");

**long** size = list.stream().skip(1)

.map(element -> element.substring(0, 3)).sorted().count();

**intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element.**

<https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html>

### **The**reduce()**Method**

**identity –**the initial value for an accumulator, or a default value if a stream is empty and there is nothing to accumulate

**accumulator –**a function which specifies the logic of the aggregation of elements. As the accumulator creates a new value for every step of reducing, the quantity of new values equals the stream's size and only the last value is useful. This is not very good for the performance.

**combiner –**a function which aggregates the results of the accumulator. We only call combiner in a parallel mode to reduce the results of accumulators from different threads.

### **The**collect()**Method**

The reduction of a stream can also be executed by another terminal operation, the collect() method. It accepts an argument of the type Collector, which specifies the mechanism of reduction. There are already created, predefined collectors for most common operations. They can be accessed with the help of the Collectors type.

In this section, we will use the following List as a source for all streams:

List<Product> productList = Arrays.asList(**new** **Product**(23, "potatoes"),

**new** **Product**(14, "orange"), **new** **Product**(13, "lemon"),

**new** **Product**(23, "bread"), **new** **Product**(13, "sugar"));

**Converting a stream to the Collection (*Collection, List*or*Set*):**

List<String> collectorCollection =

productList.stream().map(Product::getName).collect(Collectors.toList());

**Reducing to *String*:**

**String** listToString = productList.stream().map(Product::getName)

.collect(Collectors.joining(", ", "[", "]"));

The joiner() method can have from one to three parameters (delimiter, prefix, suffix). The most convenient thing about using joiner() is that the developer doesn't need to check if the stream reaches its end to apply the suffix and not to apply a delimiter. Collector will take care of that.

**Processing the average value of all numeric elements of the stream:**

**double** averagePrice = productList.stream()

.collect(Collectors.averagingInt(Product::getPrice));

**Processing the sum of all numeric elements of the stream:**

**int** summingPrice = productList.stream()

.collect(Collectors.summingInt(Product::getPrice));

The methods averagingXX(), summingXX() and summarizingXX() can work with primitives (int, long, double) and with their wrapper classes (Integer, Long, Double). One more powerful feature of these methods is providing the mapping. As a result, the developer doesn't need to use an additional map() operation before the collect() method.