Chapter-1

**State Management**

For larger Angular applications with a lot of asynchronous activity and where there's a lot of state that is being shared and manipulated across multiple components and modules, managing state can be quite challenging. In a typical application, we're managing things like:

* Data that comes from the server and whether it's pending or resulted in an

errorState Management

For larger Angular applications with a lot of asynchronous activity and where there's a lot of state that is being shared and manipulated across multiple components and modules, managing state can be quite challenging. In a typical application, we're managing things like:

Data that comes from the server and whether it's pending or resulted in an

error

UI state like toggles, alerts and errors messages

User input, such as form submissions, filters and search queries

Custom themes, credentials and localization

Many other types of state

As the application grows, how do we know that a state change in one module will consistently and accurately reflected in other modules? And what if these modifications result in even more state changes? Eventually, it becomes extremely difficult to reason about what's actually happening in your application, and be a large source of bugs.

In Angular, there are 2 main ways to solve this problem.

​;

Angular Services and RxJS.

* UI state like toggles, alerts and errors messages
* User input, such as form submissions, filters and search queries
* Custom themes, credentials and localization
* Many other types of state

As the application grows, how do we know that a state change in one module will consistently and accurately reflected in other modules? And what if these modifications result in even more state changes? Eventually, it becomes extremely difficult to reason about what's actually happening in your application, and be a large source of bugs.

In Angular, there are 2 main ways to solve this problem.

1. 1.

​[Redux using @ngrx](https://angular-training-guide.rangle.io/state-management/ngrx);

1. 2.

Angular Services and RxJS.

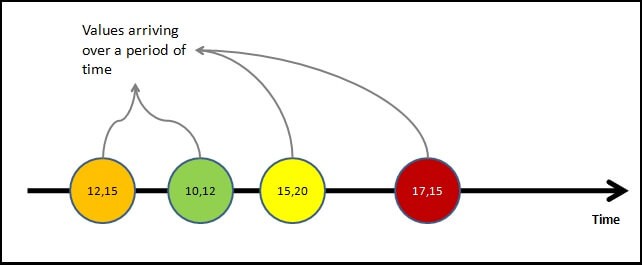
Rx stands from Reactive programming. It is defined as programming with asynchronous **data streams**. So it is important that you understand what is data stream is.

## What is a data stream

A stream is a data, which arrives over a period of time. The stream of data can be anything. Like variables, user inputs, properties, caches, data structures, and even failures, etc

Consider the example of a sequence of x & y positions of mouse click events. Assume that user has clicked on the locations (12,15), (10,12), (15,20) & (17,15) in that order.

The following diagram shows how the values arrive over a period of time. As you can see stream emits the values as they happen i.e asynchronously.

mouse click events as data streams

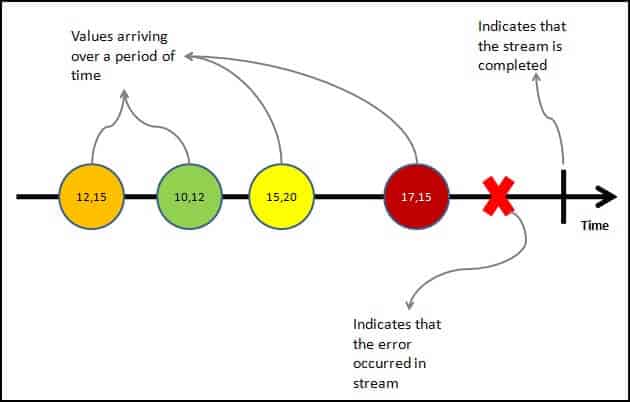
Value is not the only thing that stream emits. The stream may complete as the user closes the window or app.

* Or an error may happen which results in the closure of the stream.

At any point in time stream may emit any of the following three things

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

The following diagram shows all the three possibilities in a stream

mouse click events as data streams with emit error and complete events

As said earlier the stream of data can be anything. For Example

* Mouse click or Mouse hover events with x & y positions
* Keyboard events like keyup, keydown, keypress, etc
* Form events like value changes etc
* Data which arrives after an HTTP request
* User Notifications
* Measurements from any sensor

Important Points regarding streams can

* emit zero, one or more values of any time.
* can also emit errors.
* must emit the complete signal, when completes (finite streams).
* can be infinite, that they never complete

Now, we have understood what is a data stream, let us look at what is Reactive Programming is

## Reactive Programming

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.

This is where the RxJs comes into the picture

[The introduction to Reactive Programming you’ve been missing](https://gist.github.com/staltz/868e7e9bc2a7b8c1f754) gives you a very nice introduction to Reactive Programming. Also, refer to [Introduction to Rx](http://introtorx.com/)

## What is RxJS

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.

## features of RxJS

In RxJS, the following concepts takes care of handling the async task −

### Observable

An observable is a function that creates an observer and attaches it to the source where values are expected, for example, clicks, mouse events from a dom element or an Http request, etc.

### Observer

It is an object with next(), error() and complete() methods, that will get called when there is interaction to the with the observable i.e. the source interacts for an example button click, Http request, etc.

### Subscription

When the observable is created, to execute the observable we need to subscribe to it. It can also be used to cancel the execution.

### Operators

An operator is a pure function that takes in observable as input and the output is also an observable.

### Subject

A subject is an observable that can multicast i.e. talk to many observers. Consider a button with an event listener, the function attached to the event using addlistener is called every time the user clicks on the button similar functionality goes for subject too.

### Schedulers

A scheduler controls the execution of when the subscription has to start and notified.

## When to use RxJS?

**If your project consists of lots of async task handling than RxJS is a good choice**. It is loaded by default with the Angular project.

## Advantages of using RxJS

The following are the advantages of using RxJS −

* RxJS can be used with other Javascript libraries and frameworks. It is supported by javascript and also with typescript. Few examples are Angular, ReactJS, Vuejs, nodejs etc.
* RxJS is an awesome library when it comes to the handling of async tasks.RxJS uses observables to work with reactive programming that deals with asynchronous data calls, callbacks and event-based programs.
* RxJS offers a huge collection of operators in mathematical, transformation, filtering, utility, conditional, error handling, join categories that makes life easy when used with reactive programming.

## Disadvantages of using RxJS

The following are the disadvantages of using RxJS −

* Debugging the code with observables is little difficult.
* As you start to use Observables, you can end up your full code wrapped under the observables.

**The RxJs has two main players**

1. Observable
2. Observers ( Subscribers)

## What is an Observable in Angular

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**.

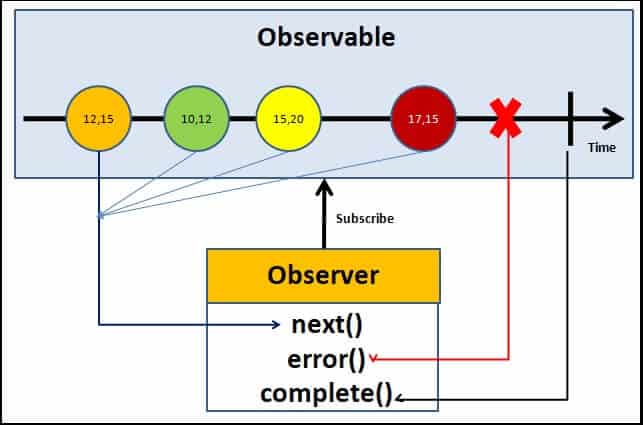
**Observable stream** or simple Observable emits the **value from the stream** asynchronously. It emits the **complete** signals when the stream completes or an **error** signal if the stream errors out.

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**.

## Who are observers (subscribers)

The Observable on its own is useless unless someone consumes the value emitted by the observable. We call them **observers or subscribers.**

The observers communicate with the Observable using callbacks

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()

Angular Observable Tutorial how observable and observers communicates with callbacks

The observable starts emitting the value as soon as the observer or consumer subscribes to it.

The observable invokes the next() callback whenever the value arrives in the stream. It passes the value as the argument to the next callback. If the error occurs, then the error() callback is invoked. It invokes the complete() callback when the stream completes.

* 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

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import {Observable} from 'rxjs';    @Component({      selector: 'app-next-example',      templateUrl: './next-example.component.html',      styleUrls: ['./next-example.component.css']  })    export class NextExampleComponent implements OnInit {        constructor() { }        ngOnInit() {            // Create a new Observable          const sqnc = new Observable(countOnetoTen);            // Execute the Observable and print the          // result of each notification          // next() is a call to countOnetoTen method          // to get the next value from the observable          sqnc.subscribe({              next(num) { console.log(num); }          });            // This function runs when subscribe()          // is called          function countOnetoTen(observer) {                for(var i = 1; i <= 10; i++) {                    // Calls the next observable                  // notification                  observer.next(i);              }                // Unsubscribe after completing              // the sequence              return {unsubscribe(){}};          }      }  } |

* **Output:**

## Angular Observable tutorial

Now, we have learned the basics of the RxJs Observable, let us now see how it works using an example.

Create a new project in angular. Remove the contents from app.component.html. Open the app.component.ts

### Import the required libraries

RxJs library is installed automatically when you create the Angular project. Hence there is no need to install it.

Import the Observable from the rxjs library

|  |  |
| --- | --- |
|  | import { Observable } from 'rxjs'; |

### Observable Creation

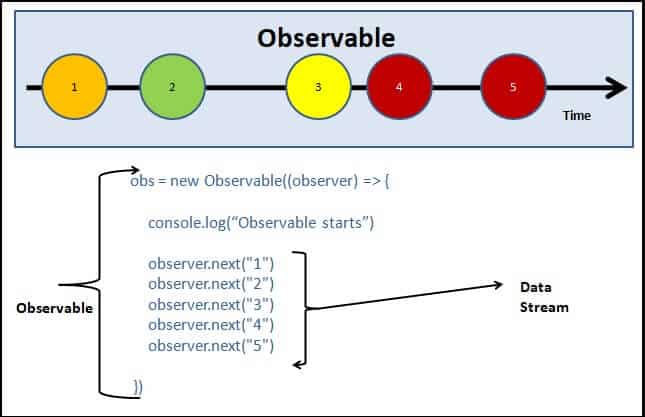
There are few ways in which you can create observable in angular. Simplest is to use the Observable constructor. The observable constructor takes observer (or subscriber) as its argument. The subscriber will run when this observable’s subscribe() method executes.

The following example creates an observable of a stream of numbers 1, 2, 3, 4, 5

|  |  |
| --- | --- |
| 10 | obs = new Observable((observer) => {       console.log("Observable starts")       observer.next("1")       observer.next("2")       observer.next("3")       observer.next("4")       observer.next("5")     }) |

The variable obs is now of the type of observable.

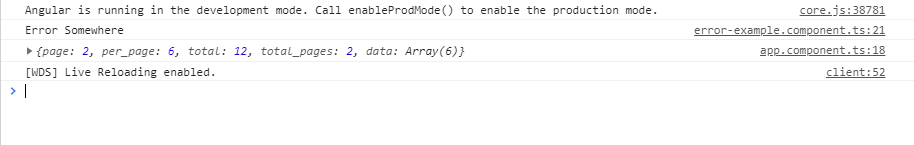
The above example declares the obs as the observable but does not instantiate it. To make the observable to emit values, we need to subscribe to it.

Creating observable in Angular Observable Tutorial app

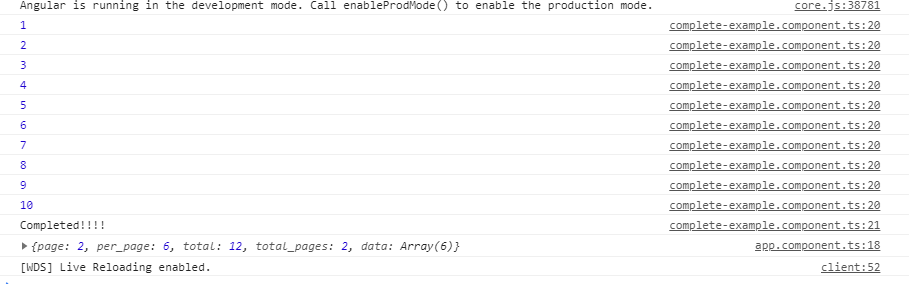
In the above example, we used the Observable Constructor to create the Observable.

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import { Observable } from 'rxjs';    @Component({      selector: 'app-error-example',      templateUrl: './error-example.component.html',      styleUrls: ['./error-example.component.css']  })    export class ErrorExampleComponent implements OnInit {        constructor() { }        ngOnInit() {            // Create a new Observable          const sqnc = new Observable(generateError);            // Execute the Observable and print the          // result of each notification          // error() is called when next generate          // some error          sqnc.subscribe({              next(num) { },              error(err) { console.log('Error Somewhere')}          });            // This function runs when subscribe() is called          function generateError(observer){                // Calls the next observable notification              // It generates an error and error is called              observer.next( adddlert("Welcome guest!"));                // Unsubscribe after completing the sequence              return {unsubscribe(){}};          }      }  **Class example**  App.component.ts  }import { Component,OnInit } from '@angular/core';  import { Observable, observable } from 'rxjs';  @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'observables';  ngOnInit(): void  {    //Called after the constructor, initializing input properties, and the first call to ngOnChanges.    //Add 'implements OnInit' to the class.    const sequence=new Observable(onetoten);    sequence.subscribe({      next(num){console.log(num);}    })  }}   function onetoten(observer:any):void  {  for (let index = 0; index <10; index++) {    observer.next(index);  }  } |

* **Output:**



* **complete:** It is a handles in which the completion of observable execution is notified.  
  **Example:** This example shows the use of the complete function. The completion notification is triggered by the observer after the completion of the execution of the **Output:**



* class example-2
* import { Component, OnInit } from '@angular/core';
* import { Observable } from 'rxjs';
* @Component({
* selector: 'app-obsex1',
* templateUrl: './obsex1.component.html',
* styleUrls: ['./obsex1.component.css']
* })
* export class Obsex1Component implements OnInit {
* constructor() { }
* ngOnInit(): void {
* const tentoone$:Observable<number>=new Observable((observer)=>{
* for (let index = 10; index >1; index--)
* {
* observer.next(index);
* }
* });
* tentoone$.subscribe(val=>console.log(val));
* }
* }

Refer this component in app.component.ts as child

Component by mentioning selector of child in

Parent template:

import { Component,OnInit } from '@angular/core';

import { ObjectUnsubscribedError, Observable, observable } from 'rxjs';

@Component({

  selector: 'app-root',

  template: `<div>

    <app-obsex1></app-obsex1>

    </div>`

  ,

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'observables';

**Making an Observable:** In the following example, we will be making a simple observable for getting a table of 2. This code is written in the app.component.ts file. Before using Observables do import Observables from rxjs library by writing the following code. 

import {Observables} from 'rxjs'

* javascript

|  |
| --- |
| import { Component } from '@angular/core';  import {Observable} from "rxjs";  import { CompileTemplateMetadata } from '@angular/compiler';    @Component({      selector: 'app-rt',      templateUrl: './app.component.html',      styleUrls: ['./app.component.css']  })    export class AppComponent {      title = 'MyFirstApp';  }    // Create a new Observable that will  // deliver the above sequence  const table = new Observable(tableOfTwo);    // Execute the Observable and print the  // result of each notification  // next() is a call to tableOfTwo() method  // to get the next value from the observable  table.subscribe({      next(num) { console.log(num); },      complete() { console.log('Finished sequence'); }  });    // This function runs when subscribe() is called  function tableOfTwo(observer) {    **for(var i = 1; i <= 10; i++) {**  **observer.next('2 \* ' + i + ' = ' + i\*2);**      }    **observer.complete();**  **return {unsubscribe(){}};**  } |

In this code, the **next()** is used to call the next return form the observable and in the observable, after the task is completed it returns unsubscribe function which leads to unsubscription of the observable and thus no more request can be undertaken. When the complete() method is called then it prints the string ‘Finished sequence’. All the functionality are been shown in the console.  
**Output:** 



**Multicasting:** It is a practice of broadcasting an observable to a list of multiple subscribers in a single execution. With multicasting observable, there are no multiple listeners to be registered on the document, but instead, the first listener is reused and the values are sent out to each subscriber. It is done by determining the observable about whether or not values are to be multicasted.  
Continuing with the above example, now multicast operation will be performed, which will cast the same table of two on two 2 sequences and will wait for a second before doing another operation.

* javascript

|  |
| --- |
| import { Component, OnInit } from '@angular/core';  import {Observable} from 'rxjs';    @Component({      selector: 'app-my-page',      templateUrl: './my-page.component.html',      styleUrls: ['./my-page.component.css']  })    export class MyPageComponent implements OnInit {        constructor() { }        ngOnInit() {            const multiSeq = new Observable(this.multiSeqSubs());              multiSeq.subscribe({              next(num) { console.log('1st subscribe: ' + num); },              complete() { console.log('1st sequence finished.'); }          });            // Subscribe again After 1 seconds.          setTimeout(() => {              multiSeq.subscribe({                  next(num) { console.log('2nd subscribe: ' + num); },                  complete() { console.log('2nd sequence finished.'); }              });          }, 1000);      }        multiSeqSubs() {          const seq = [];            for (var i = 1; i <= 10; i++) {                // Pushes the string onto sequence              seq.push('2 \* ' + i + '=' + 2\*i)          }            // Keep track of each observer          const obs = [];            // A single time Stamp for one          // set of values being generated,          // multicasted to each subscriber          let timeStamp;            // Return the subscriber function          // (runs when subscribe() function          // is invoked)          return (ob) => {              obs.push(ob);                // When this is the first subscription,              // start the sequence              if (obs.length === 1) {                  timeStamp = this.exec\_Sequence({                      next(val) {                            // Iterate through observers                          // and notify all subscriptions                          obs.forEach(o => o.next(val));                      },                        complete() {                            // Notify all complete callbacks                          obs.slice(0).forEach(o => o.complete());                      }                  }, seq, 0);              }                return {                    // Unsubscribe from the observers                  unsubscribe() {                        obs.splice(obs.indexOf(ob), 1);                        // Cleanup                      if (obs.length === 0) {                          clearTimeout(timeStamp);                      }                  }              };          };      }        // Executes the sequence      exec\_Sequence(observer, sequence, index) {          return setTimeout(() => {              observer.next(sequence[index]);              if (index === sequence.length - 1) {                  observer.complete();              } else {                  this.exec\_Sequence(observer, sequence, ++index);              }          }, 1000);      }        // Create a new Observable that will      // deliver the above sequence  } |

This code is doing the same functionality i.e. handling multicast operations. In this code, we have a list of observers that depends on the no. of subscriptions made for the multicast operation. Here during the code execution, we have only 2 operations that are executing and hence we have only 2 elements in the ‘obs’ list.  
**Output:** 



**Error Handling:**   
Observables produce asynchronous values and thus try/catch do not catch any errors because it may lead to stop the code irrespective of other tasks running at that instance of time. Instead, we handle errors by specifying an error callback on the observer. When an error is produced, it causes the observable to clean up subscriptions and stop producing values for that subscription. An observable can either produce values (calling the next callback), or it can complete, calling either the complete or error callback.

**The syntax for error callback**

observable.subscribe({

next(val) { console.log('Next: ' + val)},

error(err) { console.log('Error: ' + err)}

});

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

We subscribe to the observable, by invoking the subscribe method on it. We can optionally, include the three callbacks next(), error() & complete() as shown below

|  |  |
| --- | --- |
|  | ngOnInit() {        this.obs.subscribe(        val => { console.log(val) }, //next callback        error => { console.log("error") }, //error callback        () => { console.log("Completed") } //complete callback      )  } |

The complete app.component.ts code is as shown below.

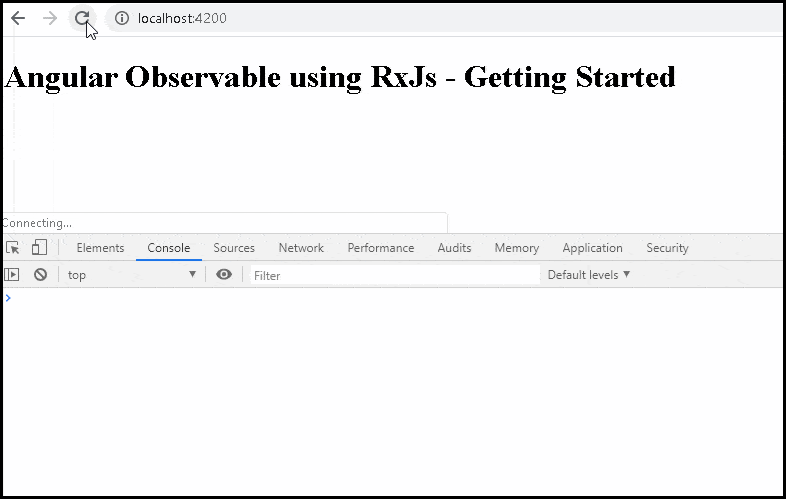
|  |  |
| --- | --- |
|  | import { Component, OnInit } from '@angular/core';  import { Observable } from 'rxjs';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {    title = 'Angular Observable using RxJs - Getting Started';      obs = new Observable((observer) => {      console.log("Observable starts")        observer.next("1")        observer.next("2")        observer.next("3")        observer.next("4")        observer.next("5")    })      data=[];      ngOnInit() {        this.obs.subscribe(        val=> { console.log(val) },        error => { console.log("error")},        () => {console.log("Completed")}      )    }  } |

Now, run the code and watch the debug window.

### Adding interval

We can add a timeout to insert a delay in each next() callback

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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);      }) |

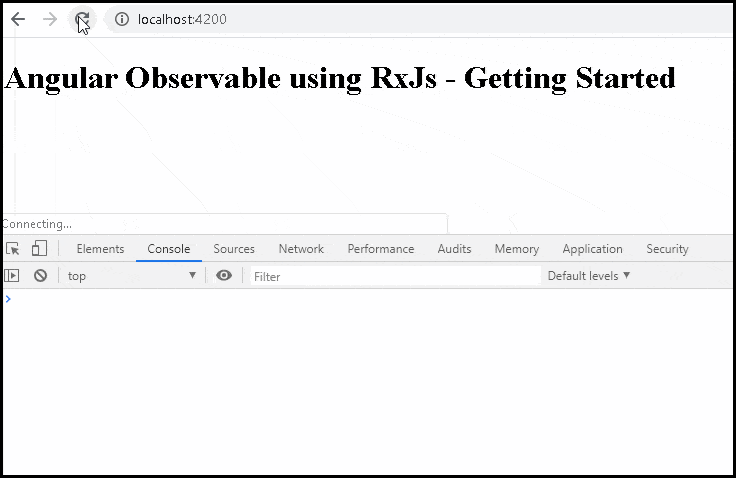
Angular Observable tutorial example app

### Error event

As mentioned earlier, the observable can also emit an error. This is done by invoking the error() callback and passing the error object. The observables stop after emitting the error signal. Hence values 4 & 5 are never emitted.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | 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);      }) |

You can send the error object as the argument to the error method

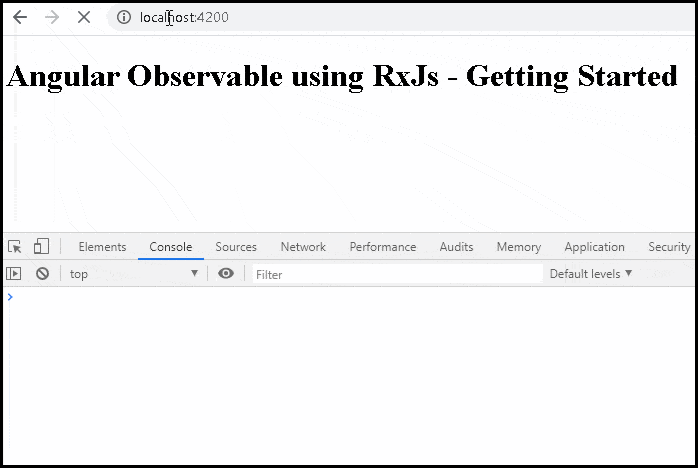
Observable with the error event

### Complete Event

Similarly the complete event. The observables stop after emitting the complete signal. Hence values 4 & 5 are never emitted.

|  |  |
| --- | --- |
|  | 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);      }) |

[***Source Code***](https://stackblitz.com/edit/angular-ivy-vbhdfs?file=src/app/app.component.ts)

Observable with complete event

## Observable Operators

The Operators are functions that operate on an Observable and return a new Observable.

The power of observable comes from the [operators](https://angular.io/guide/rx-library#operators). You can use them to manipulate the incoming observable, filter it, merge it with another observable, alter the values or subscribe to another observable.

You can also chain each operator one after the other using the [pipe](https://www.tektutorialshub.com/angular/angular-observable-pipe/). Each operator in the chain gets the observable from the previous operator. It modifies it and creates a new observable, which becomes the input for the next observable.

The following example shows the [filer](https://www.tektutorialshub.com/angular/filter-operator-in-angular-observable/) & [map](https://www.tektutorialshub.com/angular/angular-observable-map-operator/) operators chained inside a [pipe](https://www.tektutorialshub.com/angular/angular-observable-pipe/). The filter operator removes all data which is less than or equal to 2 and the map operator multiplies the value by 2.

The input stream is [1,2,3,4,5] , while the output is [6, 8, 10].

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | obs.pipe(  obs = new Observable((observer) => {      observer.next(1)      observer.next(2)      observer.next(3)      observer.next(4)      observer.next(5)      observer.complete()    }).pipe(      filter(data => data > 2),   //filter Operator      map((val) => {return val as number \* 2}), //map operator    ) |

The following table lists some of the commonly used operators

| **AREA** | **OPERATORS** |
| --- | --- |
| Combination | combineLatest, concat, merge, startWith , withLatestFrom, zip |
| Filtering | [debounceTime](https://www.tektutorialshub.com/angular/debouncetime-debounce-in-angular/), distinctUntilChanged, [filter](https://www.tektutorialshub.com/angular/filter-operator-in-angular-observable/), [take](https://www.tektutorialshub.com/angular/take-takeuntil-takewhile-takelast-in-angular-observable/), [takeUntil](https://www.tektutorialshub.com/angular/take-takeuntil-takewhile-takelast-in-angular-observable/), [takeWhile](https://www.tektutorialshub.com/angular/take-takeuntil-takewhile-takelast-in-angular-observable/), [takeLast](https://www.tektutorialshub.com/angular/take-takeuntil-takewhile-takelast-in-angular-observable/), [first](https://www.tektutorialshub.com/angular/first-last-single-operator-in-angular-observable/), [last](https://www.tektutorialshub.com/angular/first-last-single-operator-in-angular-observable/), [single](https://www.tektutorialshub.com/angular/first-last-single-operator-in-angular-observable/), [skip](https://www.tektutorialshub.com/angular/skip-skipuntil-skipwhile-skiplast-operators-in-angular/), [skipUntil](https://www.tektutorialshub.com/angular/skip-skipuntil-skipwhile-skiplast-operators-in-angular/), [skipWhile](https://www.tektutorialshub.com/angular/skip-skipuntil-skipwhile-skiplast-operators-in-angular/), [skipLast](https://www.tektutorialshub.com/angular/skip-skipuntil-skipwhile-skiplast-operators-in-angular/), |
| Transformation | bufferTime, [concatMap](https://www.tektutorialshub.com/angular/using-concatmap-in-angular/), [map](https://www.tektutorialshub.com/angular/angular-observable-map-operator/), [mergeMap](https://www.tektutorialshub.com/angular/using-mergemap-in-angular/), [scan](https://www.tektutorialshub.com/angular/the-scan-reduce-operators-in-angular/), [switchMap](https://www.tektutorialshub.com/angular/using-switchmap-in-angular/), [ExhaustMap](https://www.tektutorialshub.com/angular/using-exhaustmap-in-angular/), [reduce](https://www.tektutorialshub.com/angular/the-scan-reduce-operators-in-angular/) |
| Utility | [tap](https://www.tektutorialshub.com/angular/tap-operator-in-angular-observable/), delay, delaywhen |
| Error Handling | [throwerror](https://www.tektutorialshub.com/angular/using-throwerror-in-angular-observable/), [catcherror](https://www.tektutorialshub.com/angular/angular-catcherror/), [retry](https://www.tektutorialshub.com/angular/retry-retrywhen-in-angular-observable/), [retrywhen](https://www.tektutorialshub.com/angular/retry-retrywhen-in-angular-observable/) |
| Multicasting | share |

## Unsubscribing from an Observable

We need to [unsubscribe to close the observable](https://www.tektutorialshub.com/angular/unsubscribing-from-an-observable-in-angular/) when we no longer require it. If not it may lead to memory leak & Performance degradation.

To Unsubscribe from an observable, we need to call the Unsubscribe() method on the subscription. It will clean up all listeners and frees up the memory.

To do that, first, create a variable to store the subscription

|  |  |
| --- | --- |
| 1  2  3 | obs: Subscription; |

Assign the subscription to the obs variable

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | this.obs = this.src.subscribe(value => {    console.log("Received " + this.id);  }); |

Call the unsubscribe() method in the [ngOnDestroy](https://www.tektutorialshub.com/angular/angular-ngoninit-and-ngondestroy/) method.

|  |  |
| --- | --- |
| 1  2  3  4  5 | ngOnDestroy() {    this.obs.unsubscribe();  } |

When we destroy the component, the observable is unsubscribed and cleaned up.

But, you do not have to unsubscribe from every subscription. For Example, the observables, which emits the complete signal, close the observable.

RxJS is an impeccable framework for handling asynchronous and event-based programs; this library works on the robust observable pattern, it is also known as reactive programming. RxJS introduced to empower JavaScript.

RxJS is a collective combination of the Observer pattern, Iterator pattern on top of that functional programming with collections that help handle the events’ sequences.

**Observables in Angular**

In this angular RxJS Observable example, we use the observable interface to make the asynchronous HTTP GET request to fetch the data from the remote server.

Additionally, Observable in Angular are useful for creating custom events, and custom events help in transferring data from one component to another component.

Likewise, AJAX requests are also well synced with Observables in conjunction with HTTP modules and smoothly handle the response. Nonetheless, this detailed guide will share with you how to use observable to handle httpclient requests in angular project.

**Angular 13 Observable Asynchronous HTTP Request Example**

Here are the steps that you have to follow gradually to understand the concept of angular observable:

* **Step 1:** Install Angular Project
* **Step 2:** Register HttpClientModule
* **Step 3:** Create User Interface
* **Step 4:** Generate & Configure Angular Service
* **Step 5:** Handle HTTP Observable Response
* **Step 6:** Display Data
* **Step 7:** Test Angular App

**Install Angular Project**

Before you start with this guide, ensure you have the latest angular CLI installed on your development machine:

npm install -g @angular/cli

Bash

COPY

Thereafter, open terminal on top of that run the recommended command to install a brand new angualr app:

ng new angular-observable-http-demo

Bash

COPY

Next, move to project folder’s root:

cd angular-observable-http-demo

Bash

COPY

**Disable Strict Angular TypeStrict Errors**

The latest version of Angular comes with strict mode, you have to manually disable the strict mode you can set “strict”: false, "noImplicitReturns": false and "strictTemplates": false inside the compilerOptions and angularCompilerOptions in **tsconfig.json** file.

**Register HttpClientModule in App Module**

Open **app.module.ts** file, then import HttpClientModule from ‘@angular/common/http’ package including register the HttpClientModule in imports array.

Now, the HttpClient module is available throughout the angular components, including XSRF service support.

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

import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { HttpClientModule } from '@angular/common/http';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Create User Interface**

For enabling the type checking in angular, we need to create an interface class, hence execute the offered command:

ng generate interface user

Bash

COPY

Open and update **user.ts** interface file:

export interface User {

id: number,

name: string,

username: string,

email: string

}

**Generate & Configure Angular Service**

First, genrate the service component with reccomended command:

ng generate service user

Bash

COPY

Open **user.service.ts** file similarly update the following code:

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

import { HttpClient, HttpErrorResponse } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

import { User } from './user';

@Injectable({

providedIn: 'root'

})

export class UserService {

private API: string = 'https://jsonplaceholder.typicode.com/users';

constructor(private httpClient: HttpClient) { }

getUsers(): Observable<User[]> {

return this.httpClient.get<User[]>(`${this.API}`)

}

// Error handling

handleError(error: HttpErrorResponse) {

let errorMessage = '';

if (error.error instanceof ErrorEvent) {

errorMessage = error.error.message;

} else {

errorMessage = `Error Code: ${error.status}\nMessage: ${error.message}`;

}

console.log(errorMessage);

return throwError(errorMessage);

}

}

TypeScript

COPY

To use observable in angular, make sure to import it, moreover to handle the error gracefully also, import the throwError from rxjs package.

You can define your API in API variable, similarly import and inject HttpClient in the constructor method.

The HttpClient API provides all the options that are essential for making HTTP requests.

**Handle HTTP Observable Response in Component**

Let us check out how to incorporate angular service in TypeScript and HTML components.

Add the example code in **app.component.ts** file:

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

import { UserService } from './user.service';

import { User } from './user';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.scss']

})

export class AppComponent {

USERS = new Array<User>();

constructor(private userService: UserService) {}

ngOnInit(): void {

this.userService.getUsers().subscribe(res => {

this.USERS = res;

});

}

}

TypeScript

COPY

**Display Data**

Next, you can easily display the data from the response we received from observable HTTP.

Hecne, add the example code in **app.component.html** file:

<table class="table">

<thead>

<tr class="table-primary">

<th>#User Id</th>

<th>Name</th>

<th>User Name</th>

<th>Email</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let data of USERS">

<th scope="row">{{data.id}}</th>

<td>{{data.name}}</td>

<td>{{data.username}}</td>

<td>{{data.email}}</td>

</tr>

</tbody>

</table>

Markup

COPY

**Test Angular App**

All the work has been done, now start the application hence open the terminal use the provided command:

The above command manifest the following url, that you can use to test the app:

http://localhost:4200

**Conclusion**

The Angular HTTP Observable example tutorial is completed; in this extensive example, you understood what observable is, how to use RxJS observables to make the asynchronous HTTP request to get the data from the server in the angular app.

# ngular 13 RxJS Observable with Httpclient Example

By Hardik Savani December 15, 2021 Category : Angular

Pause

Unmute

Loaded: 1.87%

Fullscreen

Hi,

Hello all! In this article, we will talk about angular 13 observable example. We will use angular 13 rxjs observable example. This tutorial will give you simple example of angular 13 http observable example. if you want to see example of angular 13 observable with httpclient example then you are a right place. Here, Creating a basic example of observable in angular 13 example.

If you don't know how to use observable with httpclient request in angular application then i will help you getting done. we always prefer to use observable for http request that help to manage server request and monitor to server request. observable is provided by rxjs.

Here, i will give you very simple example with http request with observable in angular. we will use jsonplaceholder api to make api request. so let's follow some step to get example done, i also attach preview on bottom.

**Step 1: Create New App**

You can easily create your angular app using bellow command:

ng new my-new-app

**Step 2: Import HttpClientModule**

In this step, we need to import HttpClientModule to app.module.ts file. so let's import it as like bellow:

**src/app/app.module.ts**

**import** { BrowserModule } **from** '@angular/platform-browser';

**import** { NgModule } **from** '@angular/core';

**import** { AppComponent } **from** './app.component';

**import** { HttpClientModule } **from** '@angular/common/http';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

**export** **class** AppModule { }

Read Also: [Angular 13 Reactive Forms Validation Tutorial Example](https://www.itsolutionstuff.com/post/angular-13-reactive-forms-validation-tutorial-exampleexample.html)

**Step 3: Create Post Class**

In this step, we will simply create Post class and define data types of returning data. so let's create post.ts file and put bellow code:

**src/app/post.ts**

**export** **class** Post {

constructor(

**public** body: **string**,

**public** id: number,

**public** title: **string**,

**public** userId: number

) {}

}

**Step 4: Create Service for Call API**

Here, we need to create service for http client request. we will create service file and write client http request using observable code. this service will use in our component file. So let's create service and put bellow code:

ng g s post

Now let's add code as like bellow:

**src/app/post.service.ts**

**import** { Injectable } **from** '@angular/core';

**import** { HttpClient } **from** '@angular/common/http';

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

**import** { Post } **from** './post';

@Injectable({

providedIn: 'root'

})

**export** **class** PostService {

**private** url: **string** = 'https://jsonplaceholder.typicode.com/posts';

constructor(**private** httpClient: HttpClient) { }

**public** getPosts(): Observable<Post[]>{

**return** **this**.httpClient.**get**<Post[]>(**this**.url);

}

}

**Step 5: Use Service to Component**

Now we have to use this services to our app component. So let's updated code as like bellow:

**src/app/app.component.ts**

**import** { Component, OnInit } **from** '@angular/core';

**import** { PostService } **from** './post.service';

**import** { Post } **from** './post';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

**export** **class** AppComponent **implements** OnInit {

name = 'Angular';

posts = **new** Array<Post>();

constructor(**private** service:PostService) {}

ngOnInit() {

**this**.service.getPosts().subscribe(response => {

**this**.posts = response.map(item =>

{

**return** **new** Post(

item.body,

item.id,

item.title,

item.userId

);

});

});

}

}

**Step 6: Updated View File**

Now here, we will updated our html file. let's put bellow code:

I used bootstrap class on this form. if you want to add than then follow this link too: [Install Bootstrap 5 to Angular 13](https://www.itsolutionstuff.com/post/how-to-install-bootstrap-5-in-angular-13example.html).

**src/app/app.component.html**

**<div** **class**="container"**>**

**<h1>**Angular 13 Observables HttpClient Example - ItSolutionStuff.com**</h1>**

**<table** **class**="table table-bordered"**>**

**<tr>**

**<th>**ID**</th>**

**<th>**Body**</th>**

**<th>**Title**</th>**

**<th>**UserID**</th>**

**</tr>**

**<tr** \***ngFor**="let post of posts"**>**

**<td>**{{ post.id }}**</td>**

**<td>**{{ post.body }}**</td>**

**<td>**{{ post.title }}**</td>**

**<td>**{{ post.userId }}**</td>**

**</tr>**

**</table>**

**</div>**

**Run Angular App:**

All the required steps have been done, now you have to type the given below command and hit enter to run the Angular app:

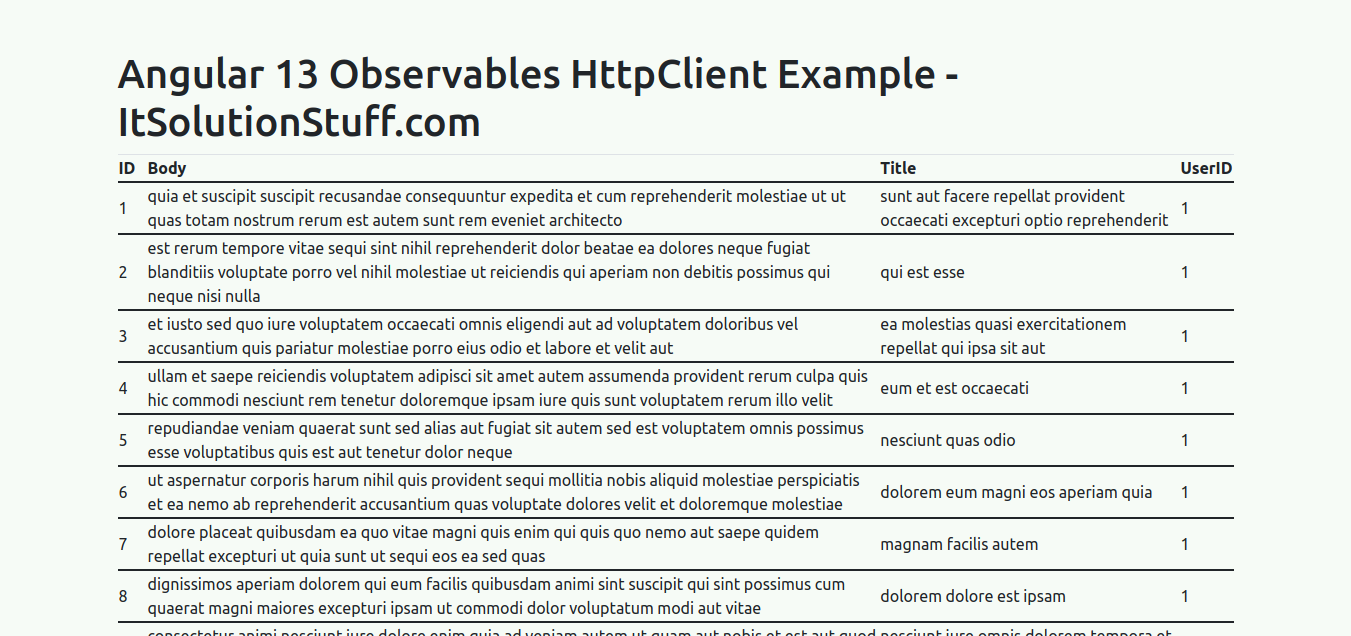
ng serve

Now, Go to your web browser, type the given URL and view the app output:

Read Also: [Angular 13 Routing Module Example Tutorial](https://www.itsolutionstuff.com/post/angular-13-routing-module-example-tutorialexample.html)

http://localhost:4200

You can simply see preview as like bellow:



* [Advertisement](https://www.itsolutionstuff.com/advertisement)

**© 2016-2021 All Rights Reserved • www.itsolutionstuff.com**

Top of Form