Reactive Programming and RxJS with Angular

.NET

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of changes in an application.

https://en.wikipedia.org/wiki/Reactive_programming

Reactive Programming – Overview

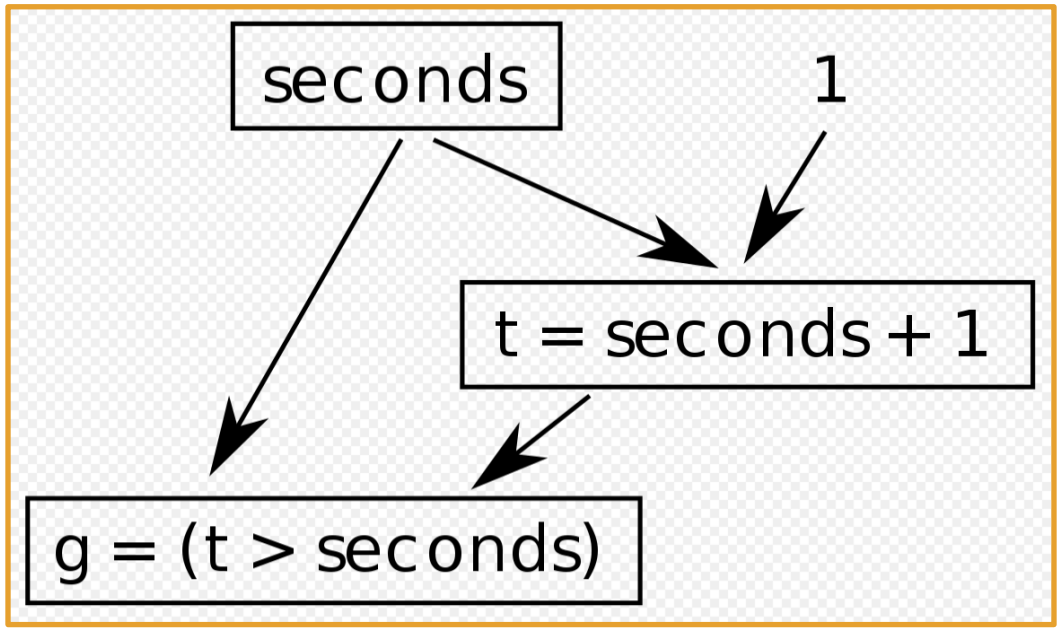
https://en.wikipedia.org/wiki/Reactive_programming
RedHatDeveloper5ThingstoKnowAboutReactiveProgramming

In ***Reactive Programming***, it is possible to create dependencies between data and to automatically propagate changes to associated values when the base value changes.

The result of an evaluation is automatically updated whenever the values of the items that make up the expression change. The program does not have to re-execute the evaluation.

If C = A + B, when the value of A or B change, C also must change. In ***Reactive Programming,*** C is automatically changed when A or B change without the program having to specifically re-evaluate the expression.

There is often a propagation graph in which a change to one value will ***propagate*** to many other values down through levels of the graph.



Reactive Programming – Challenges

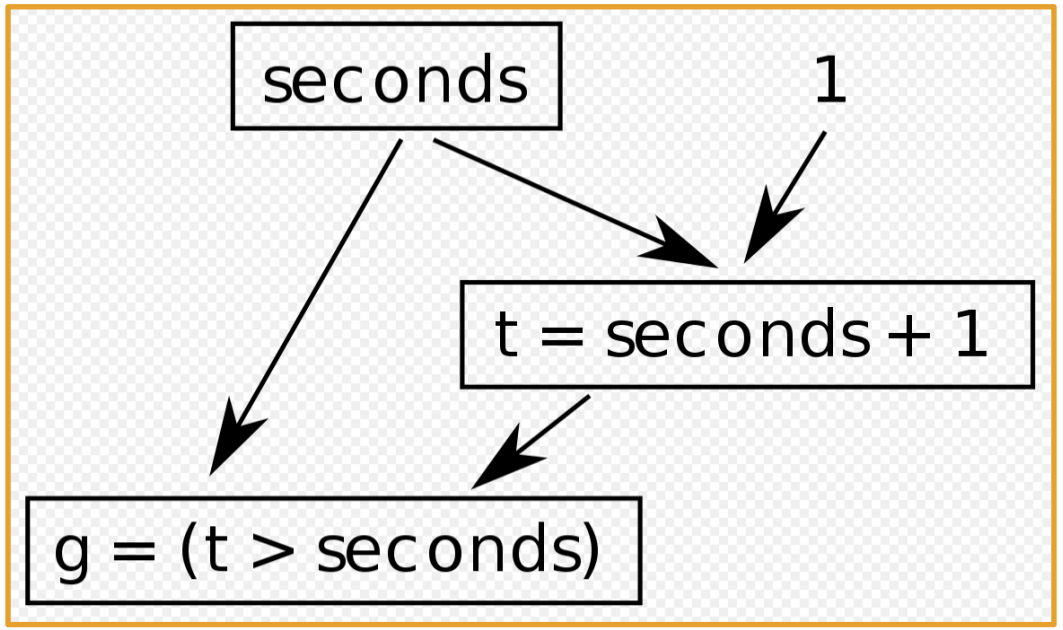
https://en.wikipedia.org/wiki/Reactive_programming
RedHatDeveloper5ThingstoKnowAboutReactiveProgramming

It is possible to have a ***propagation*** order that is not a natural consequence of the source program. In an asynchronous program, the expressions below may not execute in the same order every time.

As long as t = seconds + 1 evaluates first, then g will always evaluate to be true. But if g = (t > seconds) evaluates first (with the old value of t), then it will evaluate to be false.

Some reactive languages overcome this problem by using topological sorting expressions and updating values in topological order. This approach can delay the delivery of values.

In some cases, it is acceptable for values to be temporarily out of sync. Developers must be aware of this possibility.



Observer Pattern

https://en.wikipedia.org/wiki/Observer_pattern
https://www.learnrxjs.io/learn-rxjs/concepts/rxjs-primer

The ***Observer Pattern*** is a software design pattern in which an object, called a ***Subject***, maintains a list of its dependents, called ***Observers***, and notifies them automatically of any state changes, usually by calling their "update" method. ***Angular*** uses the ***Observer Pattern***.

The ***Observers*** are physically separated from, and have no control over, the emitted events of the ***Subject***. This pattern is for processes in which data is not available to the CPU at startup, but can arrive intermittently through HTTP requests, user input, etc.

In ***JavaScript*** (and ***TypeScript***), libraries and frameworks exist to utilize the Observer Pattern. One such library is ***RxJS***.

```
// import the fromEvent operator
import { fromEvent } from 'rxjs';

// grab button reference
const button = document.getElementById('myButton');

// create an observable of button clicks
const myObservable = fromEvent(button, 'click');

// for now, let's just log the event on each click
const subscription = myObservable.subscribe(event => console.log(event));
```

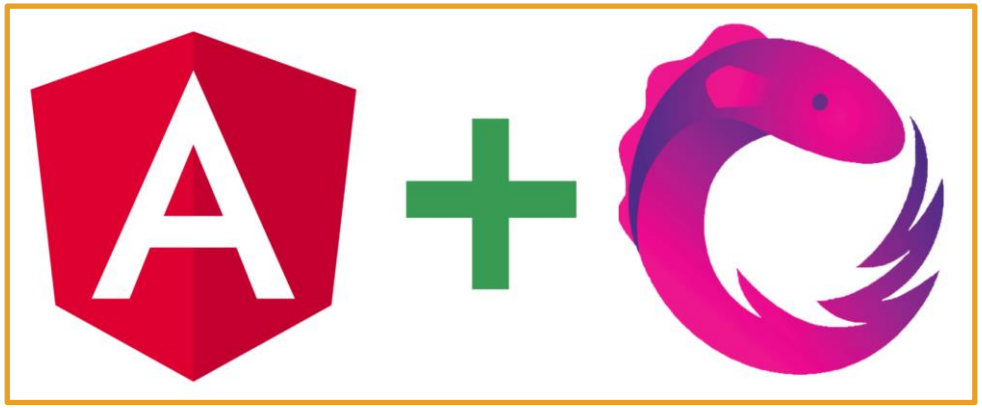
RxJS (Reactive eXtensions for JavaScript)

https://www.learnrxjs.io/
https://rxjs-dev.firebaseapp.com/guide/overview

Learning RxJS and reactive programming involves mastering a multitude of concepts and a fundamental shift in mindset from an imperative to declarative style of programming.

RxJS is a library for composing asynchronous, event-based programs by using ***observable*** sequences.

RxJS provides one core type, the Observable, satellite types (Observer, Schedulers, Subjects) and operators inspired by Array#extras (map, filter, reduce, every, etc) to allow handling of async events as collections.



Marble Diagram

https://rxjs-dev.firebaseapp.com/guide/operators

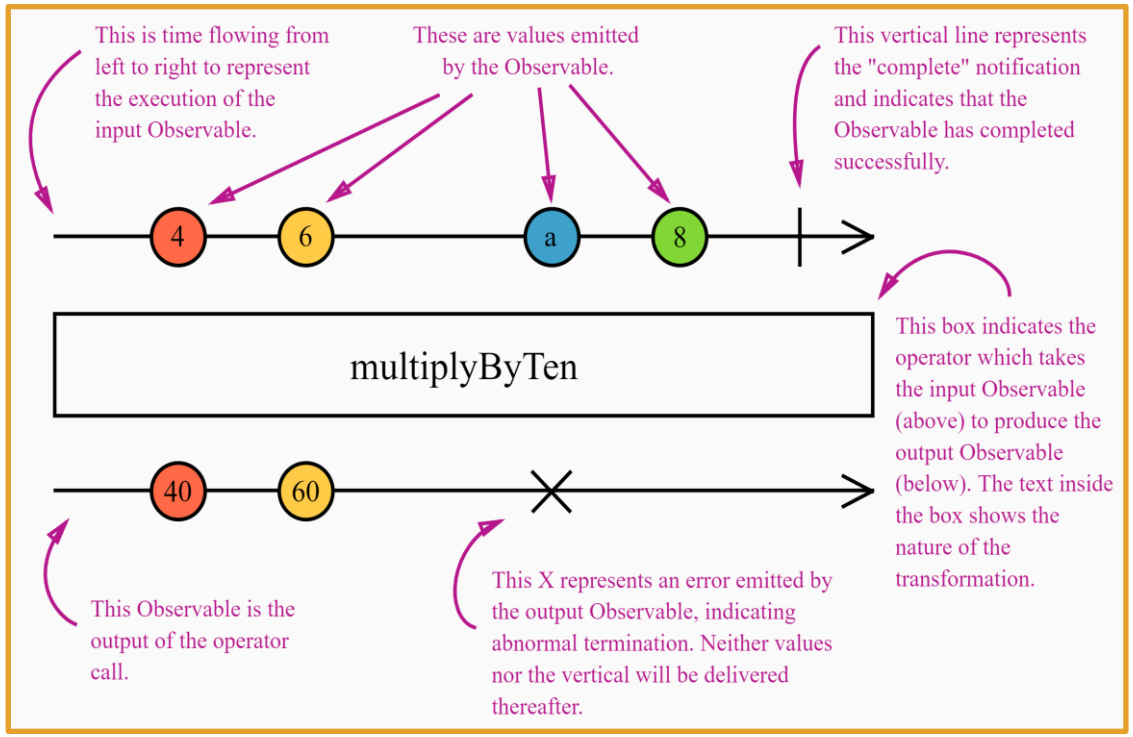
Many ***operators*** are related to time. They may delay(), sample(), throttle(), or debounce() value emissions in different ways.

Marble Diagrams are visual representations of how ***operators*** work, and include:

- the input ***Observables***,
- the ***operator*** and its parameters, and
- the output ***Observable***.

In a ***Marble Diagram***, time flows to the right, meaning the marbles on the left are older.

The diagram describes how values ("marbles") are emitted on the ***Observable*** execution.



RxJS – Core Concepts - Overview

https://www.learnrxjs.io/
https://angular.io/guide/rx-library#the-rxjs-library

Concept	Definition
Observable	Represents the idea of an invokable collection of future values or events.
Observer	A collection of ***callbacks*** that knows how to listen to values delivered by the ***Observable***.
Subscription	Represents the execution of an ***Observable***. Primarily useful for cancelling the execution.
Operators	Functions that enable a functional programming style to deal with collections (map(), filter(), concat(), reduce()).
Subject	Equivalent to an ***EventEmitter***. Used to multicast a value or event to multiple Observers.
Schedulers	Centralized dispatchers to control concurrency. Allows coordination when computation happens on ***setTimeout*** or ***requestAnimationFrame***.

RxJS - Observables

https://rxjs-dev.firebaseapp.com/guide/observable

Observables are Lazy-Push ***collections***. Lazy-Push means that the values are only updated when asked for.

An ***Observable*** deals with a ***collection*** of values or objects, while a Promise deals with just one value or object. ***Observables*** are not like ***EventEmitters*** nor are they like ***Promises*** for multiple values.

There are ***Pull*** relationships and ***Push*** relationships when it comes to receiving or sending data.

- Pull – the consumer Pulls data when it wants.
- Push – The producer (DB) Pushes data when a change is made.

	Single	Multiple
Pull	Function	Iterator
Push	Promise(Producer)	Observable(Producer)

	Producer	Consumer
Pull	Passive – sends data only when requested	Active – requests data as needed.
Push	Active – Sends data when a change occurs	Passive – reacts to data received.

RxJS - Observables

https://rxjs-dev.firebaseapp.com/guide/observable

An ***Observable*** can return multiple values over time. Functions cannot. Aside from that, subscribing to an ***Observable*** is just like calling a Function.

In this ***RxJS*** example, we:

- (Line 1)import ***Observable*** from the ***RxJS*** Library.
- (Lines 3-11)Create an ***Observable*** collection which has one parameter (with a lambda function).
- (Line 14)Upon being ***Subscribed*** to, the ***Observable*** is seeded with 1, 2, 3, waits 1 second, then adds 4 and is marked as complete.
- (Line 15-17) ***Observables*** only return:
  - next(x) iterates over the ***Observable*** values.
  - error(err) catches any error and stops further execution.
  - complete() is used for clean-up after all desired values have been iterated over

```
1.  import { Observable } from 'rxjs';
2.
3.  const observable = new Observable(subscriber => {
4.    subscriber.next(1);
5.    subscriber.next(2);
6.    subscriber.next(3);
7.    setTimeout(() => {
8.      subscriber.next(4);
9.      subscriber.complete();
10.   }, 1000);
11. });
12.
13. console.log('just before subscribe');
14. observable.subscribe({
15.   next(x) { console.log('got value ' + x); },
16.   error(err) { console.error('something wrong occurred: ' + err); },
17.   complete() { console.log('done'); }
18. });
19. console.log('just after subscribe');
```

The result of this code is:

```
just before subscribe
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

RxJS - Observables

https://rxjs-dev.firebaseapp.com/guide/observable

The ***Observable*** is Lazy-loaded. It is not initialized until it is ***Subscribed*** to (called) on line 14.

At that point it is initialized, iterated over, and its values are printed to the console.

Observables can run synchronously or asynchronously.

- function() means "give me one value synchronously"
- observable.subscribe() means "give me all values, either synchronously or asynchronously"

```
1. import { Observable } from 'rxjs';
2.
3. const observable = new Observable(subscriber => {
4.   subscriber.next(1);
5.   subscriber.next(2);
6.   subscriber.next(3);
7.   setTimeout(() => {
8.     subscriber.next(4);
9.     subscriber.complete();
10.  }, 1000);
11. });
12.
13. console.log('just before subscribe');
14. observable.subscribe({
15.   next(x) { console.log('got value ' + x); },
16.   error(err) { console.error('something wrong occurred: ' + err); },
17.   complete() { console.log('done'); }
18. });
19. console.log('just after subscribe');
```

The result of this code is:

```
just before subscribe
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

RxJS – Observers (next(), error(), complete())

https://rxjs-dev.firebaseapp.com/guide/observer

Observers are a set of three ***RxJS*** callbacks that consume the values delivered by an ***Observable***. There is one callback for each type of notification delivered by the ***Observable***: next(), error(), and complete().

You can create your own ***Observable*** and pass it to .subscribe() or you can provide them as arguments to .subscribe().

.subscribe() will instantiate an ***Observer*** object using the first callback argument as the next() observer.

Observers can be incomplete, without all three callbacks. If one of the callbacks is missing, the ***Observable*** executes normally, except some types of notifications will be ignored, because they don't have a corresponding callback in the ***Observer***.

```
const observer = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};

observable.subscribe(observer);
```

```
observable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

RxJS - Operators

https://rxjs-dev.firebaseapp.com/guide/operators
https://rxjs-dev.firebaseapp.com/api/index/function/pipe
https://angular.io/guide/rx-library#operators

RxJS is useful for its ***operators***. Even though the ***Observable*** is the foundation of ***RxJS***, ***operators*** are the essential functions that allow complex asynchronous code to be easily composed. There are two types of ***operators***.

pipeable operators –take an ***Observable*** as input and return another ***Observable***. The original ***Observable*** is unmodified.
observableInstance.pipe(operator())

```
obs.pipe(
  op1(),
  op2(),
  op3(),
  op3(),
)
```

creation operators - standalone functions to create a new ***Observable*** with some common predefined behavior or by joining other Observables.

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

map(x => x * x)(of(1, 2, 3)).subscribe((v) => console.log(`value: ${v}`));
```

```
// Logs:
// value: 1
// value: 4
// value: 9
```

RxJS – Creation Operators Examples

https://rxjs-dev.firebaseapp.com/guide/operators
https://rxjs-dev.firebaseapp.com/api/index/function/from
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield#

---

interval() takes a number (not an ***Observable***) as its input argument and produces an ***Observable*** as output:

```
import { interval } from 'rxjs';//you must import the operator

const observable = interval(1000 /* number of milliseconds */);
```

from() creates an ***Observable*** from an Array, an array-like object, a ***Promise***, an iterable object, or an ***Observable***-like object.

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

const array = [10, 20, 30];
const result = from(array);

result.subscribe(x => console.log(x));

// Logs:
// 10
// 20
// 30
```

take() specifies how many iterations on a given function to make. Here, it is combined with from().

```
import { from } from 'rxjs';
import { take } from 'rxjs/operators';

function* generateDoubles(seed) {
    let i = seed;
    while (true) {
      yield i;
      i = 2 * i; // double it
    }
}

const iterator = generateDoubles(3);
const result = from(iterator).pipe(take(10));

result.subscribe(x => console.log(x));
```

```
// Logs:
// 3
// 6
// 12
// 24
// 48
// 96
// 192
// 384
// 768
// 1536
```

RxJS – Higher Order Observables

https://rxjs-dev.firebaseapp.com/guide/operators

A "***Higher-Order Observable***" is an ***Observable*** of another ***Observable***. Here we have an ***Observable*** emitting strings that are URLs of files we want to see.

```
const fileObservable = urlObservable.pipe(

map(url => http.get(url)));
```

Work with ***Higher-Order Observables*** by "flattening" them. The concatAll() operator subscribes to each "inner" ***Observable*** that comes out of the "outer" ***Observable*** and copies all the emitted values until that ***Observable*** completes, then goes on to the next one. All the values are in that way concatenated.

```
const fileObservable = urlObservable.pipe(
  map(url => http.get(url)),
  concatAll());
```

Other ***Operators*** (called ***Join Operators***) that are useful for flattening are:

mergeAll()
switchAll()
exhaust()

RxJS - Subscription

https://rxjs-dev.firebaseapp.com/guide/subscription

A ***Subscription*** is an object that represents the execution of an ***Observable***. A ***Subscription*** has one method, .unsubscribe(), which takes no argument. It only disposes of the resource held by the ***subscription***.

Subscriptions can also be combined, so that a call to .unsubscribe() of the parent ***Subscription*** will unsubscribe .add()'ed ***Subscriptions***.

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

const observable = interval(1000);
const subscription = observable.subscribe(x => console.log(x));
// Later:
// This cancels the ongoing Observable execution which
// was started by calling subscribe with an Observer.
subscription.unsubscribe();
```

```
6.  const subscription = observable1.subscribe(x => console.log('first: ' + x));
7.  const childSubscription = observable2.subscribe(x => console.log('second: ' + x));
8.
9.  subscription.add(childSubscription);
10.
11. setTimeout(() => {
12.   // Unsubscribes BOTH subscription and childSubscription
13.   subscription.unsubscribe();
14. }, 1000);
```

RxJS - Subject

https://rxjs-dev.firebaseapp.com/guide/subject

A ***Subject*** is used for ***multicasting***. Subjects are like ***Observables*** but can multicast to many ***Observers***. Every ***Subject*** is an ***Observable***. You can .subscribe() to a ***Subject*** by providing an ***Observer*** and it will start receiving values normally.

A ***Subject*** is also an ***Observer*** object with the methods next(v), error(e), and complete().

To feed a new value to the ***Subject***, just call next(theValue), and that value will be multi-casted to the ***Observers*** registered to listen to the ***Subject***.

```
1. import { Subject } from 'rxjs';
2.
3. const subject = new Subject<number>();
4.
5. subject.subscribe({
6.   next: (v) => console.log(`observerA: ${v}`)
7. });
8. subject.subscribe({
9.   next: (v) => console.log(`observerB: ${v}`)
10. });
11.
12. subject.next(1);
13. subject.next(2);
15. // Logs:
16. // observerA: 1
17. // observerB: 1
18. // observerA: 2
19. // observerB: 2
```

RxJS - Subject

https://rxjs-dev.firebaseapp.com/guide/subject

You can also provide a ***Subject*** as the argument to the .subscribe() of any ***Observable***. subscribe() will iterate over the Argument ***Subject*** and multicast the result to all subscribed ***Observers***.

Subjects are the only way of sharing an ***Observable*** execution to multiple ***Observers***.

```
1. import { Subject, from } from 'rxjs';
2.
3. const subject = new Subject<number>();
4.
5. subject.subscribe({
6.   next: (v) => console.log(`observerA: ${v}`)
7. });
8. subject.subscribe({
9.   next: (v) => console.log(`observerB: ${v}`)
10. });
11.
12. const observable = from([1, 2, 3]);
13.
14. observable.subscribe(subject); // You can
    subscribe providing a Subject
```

```
16. // Logs:
17. // observerA: 1
18. // observerB: 1
19. // observerA: 2
20. // observerB: 2
21. // observerA: 3
22. // observerB: 3
```

RxJS - Schedulers

https://rxjs-dev.firebaseapp.com/guide/scheduler

A ***scheduler*** controls when a subscription starts and when notifications are delivered. It lets you define in what execution context an ***Observable*** will deliver notifications to its ***Observer***. It consists of three components.

Data Structure	Execution Context	Clock
A Scheduler knows how to store and queue tasks based on priority (or other criteria).	A Scheduler denotes where and when a task is executed: immediately or in another callback mechanism such as setTimeout or process.nextTick, or the animation frame).	A Schedulers virtual clock provides a notion of "time" by a getter method now(). Tasks being scheduled on a particular scheduler will adhere only to the time denoted by that clock.

RxJS – Scheduler Types

https://rxjs-dev.firebaseapp.com/guide/scheduler

When using a ***Scheduler***, you will identify one of the built-in ***Scheduler Types*** along with the ***Scheduler*** object. Each of these ***Scheduler Types*** can be created and returned by using static properties of the ***Scheduler*** object.

Scheduler Type	Usage
queueScheduler	Schedules on a queue in the current event frame (trampoline scheduler).
asapScheduler	Schedules on the micro task queue, which is the same queue used for ***promises***. Will run after the current job, but before the next job.
asyncScheduler	Schedules work with setInterval. Use this for time-based operations.
animationFrameScheduler	Used for smooth browser animations. Schedules a task that will happen just before the next browser content 'repaint'.

RxJS - Schedulers

https://rxjs-dev.firebaseapp.com/guide/scheduler

The observeOn() ***Operator*** is used to attach the ***Scheduler Type*** to the ***Observable***.

The ***Observable*** is unaffected other than having the new timing layered on top of its current output timing.

observeOn() takes 2 arguments.

- A scheduler Type.
- An (optional) delay in milliseconds.

```
1.  import { Observable, asyncScheduler } from 'rxjs';
2.  import { observeOn } from 'rxjs/operators';
3.
4.  const observable = new Observable((observer) => {
5.    observer.next(1);
6.    observer.next(2);
7.    observer.next(3);
8.    observer.complete();
9.  }).pipe(
10.   observeOn(asyncScheduler)
11. );
12.
13. console.log('just before subscribe');
14. observable.subscribe({
15.   next(x) {
16.     console.log('got value ' + x)
17.   },
18.   error(err) {
19.     console.error('something wrong occurred: ' + err);
20.   },
21.   complete() {
22.      console.log('done');
23.   }
24. });
25. console.log('just after subscribe');
```

```
//Output
just before subscribe
just after subscribe
got value 1
got value 2
got value 3
done
```

Schedulers – Step-by-Step

https://rxjs-dev.firebaseapp.com/guide/scheduler

To implement a ***Scheduler***:

1. (Ln. 1)Import the ***Operators*** and ***Schedulers*** you need.
2. (Ln 4) Create an Observable to hold the values. Normally, you would return an ***Observable*** from a service function that you
3. (Ln 9) pipe() through observeOn() before being returned, and
4. (Ln 10) add the ***Scheduler*** of your choice as the first argument to observeOn().
5. subscribe() to the Observable and provide (at least) a next() function.

```
1.  import { Observable, asyncScheduler } from 'rxjs';
2.  import { observeOn } from 'rxjs/operators';
3.
4.  const observable = new Observable((observer) => {
5.    observer.next(1);
6.    observer.next(2);
7.    observer.next(3);
8.    observer.complete();
9.  }).pipe(
10.   observeOn(asyncScheduler)
11. );
12.
13. console.log('just before subscribe');
14. observable.subscribe({
15.   next(x) {
16.     console.log('got value ' + x)
17.   },
18.   error(err) {
19.     console.error('something wrong occurred: ' + err);
20.   },
21.   complete() {
22.      console.log('done');
23.   }
24. });
25. console.log('just after subscribe');
```

```
//Output
just before subscribe
just after subscribe
got value 1
got value 2
got value 3
done
```

RxJS – fromFetch()

https://rxjs-dev.firebaseapp.com/api/fetch/fromFetch

RxJS provides an **experimental** function used to call an API over HTTP.

AbortController is required for this implementation to work and use cancellation appropriately. fromFetch() will automatically set up an internal *AbortController* in order to tear down the internal fetch() when the subscription is torn down.

If a signal is provided via the init argument, it will behave like it usually does with fromFetch().

If the provided signal aborts, the error that fetch normally rejects with (in that scenario) will be emitted as an error from the *observable*.

```
import { of } from 'rxjs';
import { fromFetch } from 'rxjs/fetch';
import { switchMap, catchError } from 'rxjs/operators';

const webSite = fromFetch('https://api.revature.com/users')
  .pipe(switchMap(response => {
    if (response.ok) { return response.json();  }
    else {
      //Server returned something other than OK. Try again.
      return of({ error: true, message: `Error ${response.status}` });
  }
 }),
 catchError(err => {
  // Network or other error, handle appropriately
  console.error(err);
  return of({ error: true, message: err.message })
  })
 );

webSite.subscribe({
  next: result => console.log(result),
  complete: () => console.log('done')
});
```

RxJS –fromFetch()

https://rxjs-dev.firebaseapp.com/api/fetch/fromFetch

Some HTTP ***responses*** use chunked transfer encoding. fromFetch() will emit a ***response*** before the body is received. If a method on the ***response*** (.text(), .json()) is called, the returned ***promise*** will not resolve until the entire body has been received.

To facilitate aborting the retrieval of ***responses*** that use chunked transfer encoding, a selector can be specified via the ***init*** (second) parameter.

```
import { of } from 'rxjs';
import { fromFetch } from 'rxjs/fetch';

//the second parameter of fromFetch() is the init parameter
const website = fromFetch('https://api.revature.com/users',
                { selector: response => response.json() });

website.subscribe({
 next: result => console.log(result),
 complete: () => console.log('done')
});
```

Step-by-step - Set-up a sample RxJS Development Environment

https://medium.com/codingthesmartway-com-blog/getting-started-with-rxjs

---

1. Create a directory for this project and move into it with:
   - mkdir rxjs-test && cd rxjs-test
2. Create a NPM package.json to hold configuration settings with: (install Node Package Manager here.)
   - npm init –y
3. Install the NPM Webpack, Webpack Web Server, TypeScript, and TypeScript Loader with:
   - npm install rxjs webpack webpack-dev-server typescript ts-loader
4. Install the Webpack Command-Line Interface with
   - npm install webpack-cli --save-dev
5. In the package.json file, add a new section. This allows you to start the program with npm run start:
   - "scripts": {
     "start": "webpack-dev-server --mode development"
     },

Step-by-step - Set-up a sample RxJS Development Environment

https://medium.com/codingthesmartway-com-blog/getting-started-with-rxjs

6. Set up Webpack. In the command line, create a new file called webpack.config.js in the root directory with:
   - touch webpack.config.js.
7. Add this to the file.

```
const path = require('path');
module.exports = {
   entry: './src/index.ts',
   devtool: 'inline-source-map',
   module: {
    rules: [ {
       test: /\.tsx?$/,
       use: 'ts-loader',
       exclude: /node_modules/
     }]
   },
   resolve: { extensions: [ '.tsx', '.ts', '.js' ] },
   output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
   }
};
```

8. Set up TypeScript. In the command line, create a new file called *tsconfig.json* in the root directory with:
   - Touch *tsconfig.json*.
9. Add this to the file (*make sure your “” are the same*):
   -

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "es6",
    "moduleResolution": "node",
    "target": "es6",
    "allowJs": true,
                  " lib " : [
      "es2017",
      "dom"
    ]
  }
}
```

Step-by-step - Set-up a sample RxJS Development Environment

https://medium.com/codingthesmartway-com-blog/getting-started-with-rxjs

10. Set up your HTML file. In the command line, create a new file called *index.html* in the root directory with:
    - Touch *index.html*.
11. Add the following to the .html file

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>RxJS Demo</title>
</head>
<body>
  <h1>RxJS Demo</h1>
  <div>
    <ul id="list"></ul>
  </div>
  <script src="/bundle.js"></script>
</body>
</html>
```

12. In the command line, create a new file called *index.ts* in the root directory with:
    - Touch *index.ts*.
13. Add this text to the *index.ts* file.

```
import { Observable } from 'rxjs';
var observable = Observable.create((observer:any) => {
  observer.next('Hello World!');
  observer.next('Hello Again!');
  observer.complete();
  observer.next('Bye');
})
observable.subscribe(
  (x:any) => logItem(x),
  (error: any) => logItem ('Error: ' + error),
  () => logItem('Completed')
);
function logItem(val:any) {
  var node = document.createElement("li");
  var textnode = document.createTextNode(val);
  node.appendChild(textnode);
  document.getElementById("list").appendChild(node);
}
```

Step-by-step - Set-up a sample RxJS Development Environment

https://medium.com/codingthesmartway-com-blog/getting-started-with-rxjs

---

14. Now you can start the sample app with:
    - npm run start
15. .