You can think of an observable as an array whose items arrive asynchronously over time. **Observables help you manage asynchronous data**, such as data coming from a backend service. Observables are used within Angular itself, including Angular’s event system and its http client service. To use observables, Angular uses a third-party library called Reactive Extensions (**RxJS**). Observables are a proposed feature for ES 2016, the next version of JavaScript.

In [Taking advantage of Observables Part one](https://blog.thoughtram.io/angular/2016/01/06/taking-advantage-of-observables-in-angular2.html) and [two](https://blog.thoughtram.io/angular/2016/01/07/taking-advantage-of-observables-in-angular2-pt2.html) we already highlighted the importance of Observables in Angular. We believe that mastering Observables can make a key difference in how we write our applications. Well, if you agree, here are some good news! This article is the first of a series of posts where we’ll explore operators of the Reactive Extensions for JavaScript (RxJS) and their practical applications.

The first operator we want to explore is the most commonly used one: map.

[**Understanding the map operator**](https://blog.thoughtram.io/angular/2016/05/16/exploring-rx-operators-map.html#understanding-the-map-operator)

We’ve probably all used map before when we were working with arrays. The idea is that each item in the collection will potentially be projected into a different value.

Here is a very simple example where an array of numbers is transformed so that each number is multiplied by 10.

let values = [1, 2, 3];

let transformed = values.map(value => value \* 10);

//prints [10, 20, 30]

console.log(transformed);

By now we may be wondering what that has to do with Observables and the map operator that is part of RxJS.

Observables are very much like arrays in a way. Well, they are actually more like Iterators but let’s not get lost in the details. The key point to understand is that both represent a sequence of values. The key difference is that with Arrays/Iterators you *pull* values out as you want to work with them whereas with Observables you get values **pushed** to you as they arrive.

It’s this similarity that allows us to take advantage of pretty much all operators that we know from the pull-based world and apply them to the push-based world.

[**map and Observables**](https://blog.thoughtram.io/angular/2016/05/16/exploring-rx-operators-map.html#map-and-observables)

Let’s start with a little demo. All we need is a simple <input> element to enter some text.

<input type="text" id="demo"/>

Then we create an Observable that emits every time that the value of our input changes.

let demoInput = document.querySelector('#demo')

let obs = Rx.Observable.fromEvent(demoInput, 'input');

// Activate the observable and log all 'pushed' events

obs.subscribe(event => console.log(event));

The payload of the Observable is the plain old Event object that is provided by the inputevent of the browser. But that may not match what we are most interested in. What if we are more interested in the current value of the input? The map operator lets us project the payload of the Observable into something else. All it takes to project the payload is this tiny change.

let obs = Rx.Observable.fromEvent(demoInput, 'input')

.map(e => e.target.value);

We can go on and chain map calls to project the data even further. For instance, it may be more convenient to work with a data structure that carries the value among with the length of the string.

let obs = Rx.Observable.fromEvent(demoInput, 'input')

.map(e => e.target.value)

.filter( value => value > 100 )

.map(v => {

return {

value: v,

length: v.length

};

});

Of course, we could have done the same in the first map call. But it’s sometimes more readable to break things into multiple steps. Notice that often we also use different operators in between of two map calls (e.g to filter something out).

If you like to play a bit with the operator yourself, here is a working demo.

At this point, you may think that Observables are really just a minor enhancement on the Observer or Promise patterns… better suited to handle a sequence of events rather than a single callback. And the .map() function certainly does not - at first glance - seem to offer any added-value. The power of Observables is revealed when you start using Rx operators to transform, combine, manipulate, and work with sequences of items emitted by Observables.

These operators allow you to compose asynchronous sequences together in a declarative manner with all the efficiency benefits of callbacks but without the drawbacks of nesting callback handlers that are typically associated with asynchronous systems.

We will see that in future articles. Watch out for the next article of this series where we’ll build upon this lesson with map() and take a look at the related flatMap operator.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Observables

*Observables are lazy collections of multiple values over time.*

yeah, right…well, actually it’s pretty easy:

1. **Observables are lazy**  
   You could think of lazy observables as newsletters. For each subscriber a new newsletter is created. They are then only send to those people, and not to anyone else.
2. **Observables can have multiple values over time**  
   Now if you keep that subscription to the newsletter open, you will get a new one every once and a while. The sender decides when you get it but all you have to do is just wait until it comes straight into your inbox.

If you come from the world of promises this is a key difference as promises always return only one value. Another thing is that observables are cancelable. If you don’t want your newsletter anymore, you unsubscribe. With promises this is different, you can’t cancel a promise. If the promise is handed to you, the process that will produce that promise’s resolution is already underway, and you generally don’t have access to prevent that promise’s resolution from executing.

### Push vs pull

A key thing to understand when using observables is that observables push. Push and pull are two different ways that describe how a data producer communicates with the data consumer.

**Pull**When pulling, the data consumer decides when it get’s data from the data producer. The producer is unaware of when data will be delivered to the consumer.

Every javascript function uses the pull. The function is a Producer of data, and the code that calls the function is consuming it by “pulling” out a single return value from its call.

**Push**When pushing, it works the other way around. The data producer (the creator of the newsletter) decides when the consumer (the subscriber to the newsletter) gets the data.

Promises are the most common way of push in JavaScript today. A promise (the producer) delivers a resolved value to registered callbacks (the consumers), but unlike functions, it is the promise which is in charge of determining precisely when that value is “pushed” to the callbacks.

Observables are a new way of pushing data in JavaScript. An observable is a Producer of multiple values, “pushing” them to subscribers.

### Observables in Angular

If you start using Angular you will probably encounter observables when setting up your HTTP requests. So let’s start there.

We have now created a simple HttpClient with a fetchUsers method that returns an observable. We probably like to display the users in some sort of list, so let’s do something with this method. Since this method returns an observable we have to subscribe to it. In Angular we can subscribe to an observable in two ways:

**Manner 1:**  
We subscribe to an observable in our template using the async pipe. The benefit of this is that Angular deals with your subscription during the lifecycle of a component. Angular will automatically subscribe and unsubscribe for you. Don’t forget to import the “CommonModule” into your module, as the async pipe will be exposed from that.

Please note the dollar sign. Using the dollar sign in the name of a variable that is an observable, is considered best practice. This way it’s easy to identify if your variable is an observable or not.

**Manner 2:**  
We subscribe to the observable ourselves using the actual subscribe()method. This can be handy if you would first like to do something with the data before displaying it. The downside is that you have to manage the subscription yourself.

As you can see the template logic is quite similar, the component logic can actually become much different en more complex if you go for manner 2. In general i would recommend to choose manner 1. As this is the most easy and you don’t have to manually manage your subscriptions. Keeping your subscriptions open while not using them is a memory leak and therefore not good.

### Creating an observable yourself

Now that you know how to deal with common observables that are given to you by Angular, it’s good to know how you create an observable yourself. The simplest version looks like this:

As you can see in the example observables are **created** by using the new Observable() call, then **subscribed** to by an observer, **executed** by calling the next() and **disposed** by calling unsubscribe().

**Creating observables**Creating observables is easy, just call the new Observable() and pass along one argument which represents the observer. Therefore i usually call it “observer” as well.

**Subscribing to observables**Remember, observables are lazy. If you don’t subscribe nothing is going to happen. It’s good to know that when you subscribe to an observer, each call of subscribe() will trigger it’s own independent setup for that given observable. Subscribe calls are not shared among multiple subscribers to the same observable.

**Executing observables**The code inside an observables represents the execution of the observables. On the parameter that was given when creating the observable there are three functions available to send data to the subscribers of the observable:

* “next”: sends any value such as Numbers, Arrays or objects to it’s subscribers.
* “error”: sends a Javascript error or exception
* “complete”: does not send any value.

Calls of the next are the most common as they actually deliver the data to it’s subscribers. During observable execution there can be an infinite calls to the observer.next(), however when observer.error() or observer.complete()is called, the execution stops and no more data will be delivered to the subscribers.

**Disposing observables**Because observable execution can run for an infinite amount of time, we need a way to stop it from executing. Since each execution is run for every subscriber it’s important to not keep subscriptions open for subscribers that don’t need data anymore, as that would mean a waste of memory and computing power.

When you subscribe to an observable, you get back a subscription, which represents the ongoing execution. Just call unsubscribe()to cancel the execution.

### Conclusion

This post should give you a better understanding of how observables actually work. To get even more out of observables the next step is to understand the power of Rxjs and all the helper functions they provide. Rxjs alone could cover multiple posts which i will write about later.