Adventures in Rx

By: Mosh Hamedani

In this guide, we'll explore other useful ways to use observables. By the end of this tutorial, you'll learn how to

- Create an observable from DOM events (the Angular way)
- Create an observable from an array
- Create an observable from a list of arguments
- Implement timers
- Run parallel operations
- Handle errors
- Get notified when an observable completes

As part of reading this tutorial, **you should type all the code snippets, run them and see the results.** This will prepare you for the upcoming quiz in this section and the coding exercise in the next section.

Do NOT copy/paste code from this document. Write all the code by hand.

Creating an observable from DOM events

Earlier, you learned how to create an observable from DOM events:

```
var element = $(“#search”);
var observable = Observable.fromEvent(element, “keyup”);
```

This code is tightly coupled to the DOM and is hard to unit test. Client-side code coupled to the DOM is as bad as server-side code that talks directly to a database without a set of abstractions.

In Angular apps, we should never use libraries like jQuery or work directly with the **document** object! The whole point of Angular components is to decouple our code from the DOM. So, we use property binding and let Angular work with the DOM instead.

The reason I used jQuery in the videos was to strip away the unnecessary complexity, so you could focus on the observables and their benefits.

The Angular way

In the section about forms, we looked at the **Control** class: it represents an input field in a form. This class has a property called **valueChanges**, which returns an observable. We can subscribe to this observable and get notified as the value in the input field changes. This way, we will not be working directly with a DOM element.

1- Use the following template in **AppComponent**

```
<form [ngFormModel]="form">
    <input type="text" ngControl="search">
</form>
```

2- Create the **form** object in **AppComponent**:

```
import {ControlGroup, FormBuilder} from 'angular2/common';


form: ControlGroup;


constructor(fb: FormBuilder) {
     this.form = fb.group({
          search: []
     });
}
```

3- Get a reference to the “search” control and subscribe to its **valueChanges** property (in the constructor):

```
var search = this.form.find('search');
search.valueChanges
      .subscribe(x => console.log(x));
```

4- Run the application and type something into the input field. Note that with every keystroke, **valueChanges** observable pushes the current value of the input field.

The **valueChanges** property is also available in the **ControlGroup** class. We can use this to be notified when anything changes in a control group (or in the entire form).

5- Apply the following operators before subscribing to valueChanges. What will you see in the console?

```
     .debounceTime(400)
     .map(str => (<string>str).replace(' ', '-'))
```

Note that **<string>str** casts **str** (which is of type “any”) to a string.

Creating an observable from an array

Creating observables is not limited to asynchronous data streams. We can create an observable from an array:

```
var observable = Observable.fromArray([1, 2, 3]);
```

This line will return an observable with three items: 1, 2 and 3. So, when subscribing to this observable, we’ll get three values pushed at us.

What is the benefit of converting an array to an observable stream? We can tie this observable stream with another observable created off DOM events, AJAX calls, web sockets, etc. For example, think of a travel search engine like Kayak. Many travel search engines let the user search for flights for exact dates or within a 2-day window.

1- Let’s say the user selects the 2-day window. Create an array of the travel start dates with the 2-day window:

```
var startDates = [];
var startDate = new Date(); // Assuming today for simplicity

for (var day = -2; day <= 2; day++) {
    var date = new Date(
        startDate.getFullYear(),
        startDate.getMonth(),
        startDate.getDate() + day);
```

```
        startDates.push(date);
}
```

2- Convert this array to an observable, and for each data element, use the map operator to call the server and get the deals for the given date:

```
Observable
    .fromArray(startDates)
    .map(date => console.log("Getting deals for date " + date))
    .subscribe(x => console.log(x));
```

What do you see in the console? Why do we get “undefined” after each call to the server? Because our map operator is not returning any values to the subscribers.

3- Modify the map operator as follows:

```
    .map(date => {
           console.log("Getting deals for date " + date);
           return [1, 2, 3];
        })
```

Here, [1, 2, 3] simulates the flight deals returned from the server for the given date.

Note: For simplicity, I excluded the end date as part of getting the deals.

Other ways to create an observable

We can use the static **Observable.of()** method to create an observable from a list of arguments:

```
// Returns an observable with one item: 1
Observable.of(1);


// Returns an observable with three items: 1, 2, 3
Observable.of(1, 2, 3);


// Returns an observable with one item: [1, 2, 3]
Observable.of([1, 2, 3]);
```

Given the following code snippet:

```
var observable = …
observable.subscribe(x => console.log(x));
```

Explore different ways to create an observable. Make sure you understand how each method works:

```
Observable.empty()
Observable.range(1, 5)
Observable.fromArray([1, 2, 3])
Observable.of([1, 2, 3])
```

Implementing a timer

We can use the static **Observable.interval()** method to create a timer. This is useful for running an asynchronous operation at specified intervals.

1- Create an observable using the interval method:

```
var observable = Observable.interval(1000);
observable.subscribe(x => console.log(x));
```

You'll see a number in the console every 1000 milliseconds. This number is just a zero-based index and doesn't have much value on its own. But we can extend this example and build a client that calls the server once a minute (rather than every second) to get the latest tweets, emails, news feed, etc.

2- Apply the map operator and see the results in the console:

```
observable
    .map(x => {
        console.log("calling the server to get the latest news");
    })
    .subscribe(news => console.log(news));
```

Note that I've also changed the name of the argument of the arrow function in the subscribe method from **x** to **news**.

So, you'll see a message in the console simulating a call to the server every 1000 milliseconds. This is similar to using setInterval() method of Javascript.

Note that our **map** operator doesn't return any values and that's why we get "undefined" in the console.

3- Modify the map operator as follows:

```
.map(x => {
    console.log("calling the server to get the latest news");
    return [1, 2, 3];
})
```

Here we are assuming that [1, 2, 3] are the latest news/tweets/messages since the last time we polled the server.

4- As you've learned, we can create an observable for AJAX calls. Let's simulate an AJAX call by modifying the **map** operator to return an observable from our array. This means we should also replace **map** with **flatMap**, otherwise, we'll end up with an Observable<Observable<number[]>> (observable of observable of number[]).

```
.flatMap(x => {
    console.log("calling the server to get the latest news");
    return observable.of([1, 2, 3]);
})
```

Running Parallel Operations

Let's say you want to build a page and display the user's profile data and their tweets. Assume that your RESTful API, by design, does not return the user's profile and their tweets in a combined format. So you need to send two requests in parallel: one to get the profile, and another to get the tweets.

We can use the **forkJoin** operator to run all observables in parallel and collect their last elements.

```
Observable.forkJoin(obs1, obs2, …)
```

1- Create two observables, one for the user, one for their tweets.

```
var userStream = Observable.of({
      userId: 1, username: 'mosh'
}).delay(2000);

var tweetsStream = Observable.of([1, 2, 3]).delay(1500);
```

I've used the **delay** operator here to simulate an AJAX call that takes 1500/2000 milliseconds.

2- Use **forkJoin** to run both these observables in parallel. Note the result in the console.

```
Observable
    .forkJoin(userStream, tweetsStream)
    .subscribe(result => console.log(result));
```

3- So, **forkJoin** returns an array of data elements collected from multiple observables. Use the **map** operator to map this array into a data structure that our application expects:

```
Observable
    .forkJoin(userStream, tweetsStream)
    .map(joined =>
```

```
            new Object({user: joined[0], tweets: joined[1] }))
    .subscribe(result => console.log(result));
```

Handling Errors

Asynchronous operations that involve accessing the network can fail due to request timeouts, server failures, etc. We can handle such failures by providing a callback function as the second argument to the **subscribe** method:

```
observable.subscribe(
   x => console.log(x),
   error => console.error(error)
);
```

1- Simulate a failed AJAX call, by creating an observable using the static **Observable.throw()** method. This method returns an observable that terminates with an exception.

```
var observable = Observable.throw(new Error("Something
failed."));
```

2- Subscribe to this observable and provide an error handler:

```
observable.subscribe(
    x => console.log(x),
    error => console.error(error)
);
```

Note the result in the console.

Retrying

We can retry a failed observable using the **retry** operator.

```
observable.retry(3)
```

```
// Retry indefinitely
observable.retry()
```

1- Simulate an AJAX call that fails twice and succeeds the third time.

```
var counter = 0;
```

```
var ajaxCall = Observable.of('url')
    .flatMap(() => {
        if (++counter < 2)
           return Observable.throw(new Error("Request failed"));

        return Observable.of([1, 2, 3]);
    });
```

So, here, **counter** is just a local variable we're using for simulating an AJAX call that fails the first two times. The third time it returns the array [1, 2, 3].

In a real-world application, you wouldn't create an observable for an AJAX call like this. In Angular, we have a class called **Http,** which we use for making AJAX calls. You'll learn about this class in the next section. All methods of the **Http** class return an observable.

2- Subscribe to this observable and note the result in the console:

```
ajaxCall
    .subscribe(
        x => console.log(x),
        error => console.error(error)
);
```

3- Use the **retry** operator before subscribing and note the difference:

```
ajaxCall
    .retry(3)
    .subscribe(
         x => console.log(x),
         error => console.error(error));
```

Imagine if you had to implement the retry imperatively (using logic). At a minimum, you would need a loop and a conditional statement. With observables, it's just a matter of calling the **retry()** method. So, observables makes dealing with asynchronous operators easier.

Catching and continuing

We can use the **catch** operator to continue an observable that is terminated by an exception with the next observable.

For example, in a real-world application, we may call a remote service to get some data, but if the call fails for any reason, we may want to get the data from local storage or present some default data to the user. We can create another observable from an array and use the **catch** operator to continue the first failed observable with the second observable.

1- Run the following code snippet and see the result:

```
var remoteDataStream = Observable.throw(new Error("Something
failed."));


remoteDataStream
    .catch(err => {
        var localDataStream = Observable.of([1, 2, 3]);
        return localDataStream;
    })
    .subscribe(x => console.log(x));
```

So, if **remoteDataStream** throws an exception, we'll catch it and then return another observable (**localDataStream**).

2- Modify the creation of **remoteDataStream** as follows:

```
var remoteDataStream = Observable.of([4, 5, 6]);
```

Do you see the difference? So, the **catch** operator is only called if the first observable fails.

Timeouts

What if we call a remote service that doesn't always respond in a timely fashion? We don't want to keep the user waiting. We can use the **timeout** operator:

```
// times out after 200 milliseconds.
observable.timeout(200);
```

1- Simulate a long running operation:

```
var remoteDataStream = Observable.of([1, 2, 3]).delay(5000);
```

2- Use the **timeout** operator and then subscribe to this observable:

```
remoteDataStream
    .timeout(1000)
    .subscribe(
        x => console.log(x),
        error => console.error(error)
    );
```

With this, the observable times out after 1 second and you'll get the error message in the console.

Getting notified when an observable completes

Observables can go in the "completed" state, which means they will no longer push data items in the future. An observable wrapping an AJAX call is an example of an observable that completes. If an AJAX calls succeeds, the response will be placed in the observable and pushed at the subscriber. And if it fails, the observable will terminate with an exception. Either way, the observable completes and it will no longer emit any values or errors.

We can get notified when an observable completes by providing a third argument to the subscribe method:

```
observable.subscribe(
     x => console.log(x),
     error => console.log(error),
     () => console.log("completed"));
```

Use this **subscribe** method with the following observables and note the messages logged in the console:

```
Observable.throw(new Error("error"))
Observable.fromArray([1, 2, 3])
```

Summary

In this guide, you learned about:

- The **valueChanges** property of **Control** and **ControlGroup** objects. This property is an observable that we can subscribe to, to get notified when the value of an input field changes.

- Various ways to create an observable

```
Observable.fromArray([1, 2, 3])
Observable.of(1)
Observable.of(1, 2, 3)
Observable.of([1, 2, 3])
Observable.empty()
Observable.range(1, 3)
```

- Creating timers using **Observable.interval()**

```
Observable.interval(1000)
    .map(…)
```

- Running parallel operations using **Observable.forkJoin()**

```
Observable.forkJoin(obs1, obs2)
```

- **Observable.throw()** to return an observable that terminates with an exception.

- Handling errors by supplying a callback to the **subscribe()** method

```
obs.subscribe(
    x => console.log(x),
    error => console.error(x)
);
```

- Retrying

```
 obs.retry(3).subscribe()
```

- Catching and continuing

```
obs.catch(error => {
      return anotherObservable;
})
```

- Using timeouts:

```
obs.timeout(1000).subscribe()
```

- Getting notified when an observable completes:

```
obs.subscribe(
    x => console.log(x),
    error => console.error(error),
    () => console.log("completed")
)
```