**Introduction to RxJS**

Let’s learn about RxJS.

**We'll cover the following**

* [What is RxJS?](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4886319727116288#What-is-RxJS)
* [Why RxJS?](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4886319727116288#Why-RxJS)

In the last project we worked on, we saw a line of code that’s still somewhat of a mystery. I’m referring to this line in the app.component.ts file:

this.cocktail.search(query).subscribe((response: any) => {  
  this.drinks = response.drinks;  
});

What does the subscribe() function do? We know that it provides us with the response from the request, but it’s not exactly clear as to how it works. Under the hood, Angular is using something called RxJS.

**What is RxJS?**

RxJS is a separate library from Angular. The team behind Angular liked RxJS so much that they decided to integrate it into the framework. It’s a completely independent library. More information about the library can be found [here](https://rxjs.dev/).

The purpose of RxJS is to make dealing with asynchronous tasks easier. You can think of it as an alternative to promises and async/await syntax in JavaScript. With that being said, you don’t have to use RxJS. It’s completely optional. If you prefer promises or async/await, then, by all means, use it.

**Why RxJS?**

There are two main reasons why using RxJS is encouraged. First, it makes data more manageable. We’ll be able to see the flow of data with RxJS. Second, it’s much easier to test. We’ll look at some examples soon to better understand these points.

One of the biggest hurdles of learning Angular is learning RxJS. It’s arguably the hardest thing to learn. If you can master RxJS, then learning the rest of Angular will be a walk in the park.

**Observables**

In this lesson, we'll learn about Observables in RxJS

**We'll cover the following**

* [Terminology](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5465091601858560#Terminology)
* [Creating an observable](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5465091601858560#Creating-an-observable)

We’ll start from the basics and work our way up. We’ll be stepping away from our project for this example. Once we have a good understanding of RxJS, we’ll look at how this ties into our application.

**Terminology**

Half the battle of learning RxJS is understanding the terminology. The first term we’ll learn is **observable**. According to the dictionary, the word **observe** means to watch something. Therefore, we can conclude that an **observable** is the thing we’re watching.

This leads us to further questions. What are we watching, and why?

Concerning RxJS, an observable is an object. When we create an observable, we’re watching an object. What exactly are we watching for? Values! An observable is an object that emits values. It is the *source of values*.

**Creating an observable**

Let’s create an observable with RxJS. For this example, we’ll create an <input> element to listen for the input event. In Vanilla JavaScript, we would do the following:

const input = document.querySelector('input');  
  
input.addEventListener('input', (event) => {  
  *// Some code here...*  
});

In RxJS, we’ll do the following.

**OutputJavaScriptHTML**

const { fromEvent } = rxjs;

const input = document.querySelector('input');

const observable = fromEvent(input, 'input');

Save

Reset

There are a couple of things going on. First, in the HTML document, we have an input and we’re loading a script. We’re loading RxJS from a CDN. In our projects, RxJS is downloaded as a dependency. Just to keep things simple, we’ll use a CDN.

const { fromEvent } = rxjs;

In the script file, we’re destructuring the fromEvent function from the rxjs object. The rxjs object is an object defined by the script we loaded. The fromEvent function will help us create an observable.

It’s common practice to destructure the object to grab the functions we want. This allows for tree shaking thus reducing the size of our bundle. It won’t apply in this case because we’re using a CDN, but it’s what we’ll do when going back to working in an actual project.

The fromEvent() function has two arguments. The first is the target on which we want to listen to an event. In this case, we want to listen for an event on the <input> element. The second argument is the name of the event to which we’ll be listening.

This will return an observable. We’re observing the <input> element for the input event. The observable is stored in a variable called observable.

This line:

fromEvent(input, 'input');

is the same as this line:

input.addEventListener('input');

It’s just a difference in syntax. We’ve successfully created an observable. At the moment, nothing will happen because we aren’t handling the input. We are just observing the input for a value.

In the next couple of lessons, we’ll start handling the value.

**Observers**

Learn how to manipulate the data with pipes and operators.

We have an observable, but we’re not doing anything with it. Observables can emit values. If we want to listen for when a value is emitted, we need to create an **observer**.

Observers are the technical term for an object that listens to values delivered by the observable. We can create an observer using the subscribe() function. The observer’s job is to read the value or handle the error if one comes out.

Let’s look at how one is made.

**OutputJavaScriptHTML**

const { fromEvent } = rxjs;

const input = document.querySelector('input');

const observable = fromEvent(input, 'input');

observable.subscribe({

    next(value) {

        console.log(value);

    }

})

Save

Reset

Console

Clear

In this example, we’re calling the subscribe() function on the observable. This is how we create an observer. We’re passing an object where we have one function called next(). The object we pass in is what’s called the observer.

There are three possible functions we can define in the object: next(), error(), and complete() functions.

The next() function will run whenever the observable pushes out a new value. The error() function runs when there is an error. The complete() function runs when the observable will no longer push new values. In most cases, you’ll use the next() function. We aren’t defining the other functions because we don’t need them at the moment.

In the next() function, we have one parameter, which is the value emitted from the observable. We’re logging the value in the console.

If we run the example, we’ll see the event object logged in the console. From here we can do whatever we want with the value.

**Pipes and Operators**

Learn how to use pipes and operators to modify the value emitted from an observable.

**We'll cover the following**

* [Understanding operators](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6520076498370560#Understanding-operators)
* [Using an operator](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6520076498370560#Using-an-operator)
* [Multiple operators](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6520076498370560#Multiple-operators)

In some cases, we may not want to directly deal with the values from an observable. We may need to filter and transform it before delivering it to the observer. RxJS allows us to transform an observable by using pipes and operators.

**Understanding operators**

An operator is a technical term for a function. Plain and simple. An operator takes in an observable, transforms it, and spits out a brand new observable. We can use an operator to alter the value.

**Using an operator**

The current observable will emit the event object. This is because we’re listening for the input event on the <input> element. Let’s say rather than returning the entire event object, we just want to return the value in the input.

We can use an operator to change how the observable initially behaves. Let’s update our code to the following:

**OutputJavaScriptHTML**

const { fromEvent } = rxjs;

const { map } = rxjs.operators;

const input = document.querySelector('input');

const observable = fromEvent(input, 'input')

    .pipe(

        map(event => event.target.value)

    );

observable.subscribe({

    next(value) {

        console.log(value);

    }

})

Save

Reset

Console

Clear

In the example above, we’re chaining the pipe() function on the observable. The pipe() function allows us to group multiple operators.

**Why multiple operators?**

Good question! It’s because this allows us to swap operators whenever we need to. Technically, we can stick to using a single operator, but there are some downsides to doing so. In some situations, we may want to take a different approach as to how a value is updated. By using one operator, it becomes difficult to do so. We’ll see an example of how multiple operators work later in the section.

It’s common to change operators because RxJS comes with **over 100 operators**. You’ll soon discover that there are some operators better suited for performing a task than others.

We won’t be going over every operator. It would be too overwhelming to do so. We’ll focus on some of the most common ones.

You can think of a pipe() function as creating a pipeline of operators. It will take care of passing the value from one operator to the next.

The first operator we’re using is called map(). Above, we’re destructuring the rxjs.operators object for the map() function. This is for comprehensibility.

In the pipe() function, we’re passing in the map() function. The map() function will go through a collection and return a modified version of that collection. In this instance, we’ll be able to access the event object. We’re returning the event.target.value property. Any observer that subscribes to the observable will receive the value from the input instead of the entire event object.

The result of all this will be a new observable. If we run the example, we’ll see the value from the input get logged.

**Multiple operators**

The pipe() function will accept an unlimited number of operators. The operators will run in the order they’re passed. Each operator will return a new value.

You can imagine this as a pipeline of operators. Whatever gets emitted from an observable will flow through the pipeline.

Let’s look at how to create multiple operators.

**OutputJavaScriptHTML**

const { fromEvent } = rxjs;

const { map } = rxjs.operators;

const input = document.querySelector('input');

const observable = fromEvent(input, 'input')

    .pipe(

        map(event => event.target.value),

        map(value => parseInt(value))

    );

observable.subscribe({

····next(value)·{

········console.log(value);

····}

})·

Save

Reset

Console

Clear

We have a second operator that’s converting the value into an integer with the parseInt() function. The value from the event.target.value property will always be a string, even if we input a numeric value. We’re using the parseInt() function to change the type to an integer.

To recap, operators are functions that can transform a value in an observable. They’ll return a new observable. Pipes are a way to group multiple operators. The pipe() function will take care of passing the observable from one operator to the next.

**Avoiding Single Operators**

Let’s take a look at why we should avoid using a single operator.

**We'll cover the following**

* [The pluck operator](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5496185017597952#The-pluck-operator)
* [Using multiple operators vs. a single operator](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5496185017597952#Using-multiple-operators-vs-a-single-operator)

In the last lesson, we used two operators: one for retrieving the value from the input and another for parsing it as an integer. If we wanted to, we could have put everything into a single operator, as shown here:

map(event => parseInt(event.target.value)

However, this is not recommended because we won’t be able to swap operators if we ever need to. If we do, we’ll have to change the entire chain. Let’s look at an example.

**The pluck operator**

Let’s update our code to the following:

**OutputJavaScriptHTML**

const { fromEvent } = rxjs;

const·{·map,·pluck·}·=·rxjs.operators;

const input = document.querySelector('input');

const observable = fromEvent(input, 'input')

    .pipe(

        pluck('target', 'value'),

        map(value => parseInt(value))

    );

observable.subscribe({

    next(value) {

        console.log(value);

    }

})

Save

Reset

Console

Clear

We’re introducing a new operator called pluck(). We’re destructuring it from the rxjs.operators object, where all the other operators are defined. The pluck() function can be used to retrieve a value from an object.

That’s exactly what we were doing with the first map() function in the pipeline. The pluck() function is much more suited for this task since it’s sole purpose is to retrieve a value from an object. We want our operators to be as specific as possible. While the map() function is excellent, it isn’t clear as to what it does until we look inside. One look at the pluck() function, and we’ll instantly know what’s going on.

The pluck() function has an infinite number of arguments. It’s the property we’d like to retrieve from the object. If we have a property nested deeply in the object, like in our case, then we can pass in a chain of properties to look into. We’re passing in target and value. The pluck() function will search for a property called target and then search for a property inside the target object called value. This is what’s returned by the pluck() function:

If we run the example, we’ll be able to see the app still working. The number gets logged in the console.

**Using multiple operators vs. a single operator**

By using multiple operators, we can swap them out easily without disturbing the rest of the chain. There are over 100 different operators. As you continue to use RxJS, you’ll find alternative operators that are more suited for the job than what you were previously using. Here’s a full [list](https://rxjs.dev/api?type=function).

If we were to use a single operator, we wouldn’t be able to use another operator. We’d be bound to a single one. Imagine if we had a much more complex example. If another developer were to look at our code, it would take some time to figure out what’s going on.

In most cases, you’ll want to chain multiple operators together for readability.

**Note:** Entering a non-numeric value will result in the console outputting NaN.

**Handling Errors**

In this lesson, we'll look at how we can handle errors in RxJS.

Error handling in RxJS is simple. There are a couple of things to know. First, if an error occurs, RxJS will stop the flow of data in the pipeline. If an operator throws an error, any subsequent operators will not run. Instead, the observable will tell any observers that an error has occurred. From there, the observer can handle the error.

Let’s try throwing an error.

const observable = fromEvent(input, 'input')

    .pipe(

        pluck('target', 'value'),

        map(value => parseInt(value)),

        map(value => {

          if(isNaN(value)){

            throw new Error('Not a number!');

          }

          return value;

        })

    );

In the example above, we’re adding another map() function to the pipeline. Inside it, we’re checking if the value parameter is a valid number after it runs through the parseInt() function in the previous operator. If isNaN() returns true, we’ll throw an error. Otherwise, we’ll return the value.

When it comes to errors, observables are responsible for throwing errors. Handling errors should be performed in the observer. Let’s update the observer to handle the error.

const { fromEvent } = rxjs;

const { map, pluck } = rxjs.operators;

const input = document.querySelector('input');

const observable = fromEvent(input, 'input')

    .pipe(

        pluck('target', 'value'),

        map(value => parseInt(value)),

        map(value => {

          if(isNaN(value)){

            throw new Error('Not a number!');

          }

          return value;

        })

    );

observable.subscribe({

    next(value) {

        console.log(value);

    },

    error(err) {

        console.log(err.message);

    }

})

We aren’t going to do anything drastic with the error. In the object we passed into the subscribe() function, we defined a function called error(). If it’s defined, RxJS will run the function. In our function, we have a parameter called err. RxJS will provide our function with an object with information about the error. We’re logging the message property to view the error.

If we run the example, the console will log the value if we input a number. Anything other than a number will result in the output of an error message.

**Creating Raw Observables**

In this lesson, we'll look at how we can create an observable from scratch.

**We'll cover the following**

* [The Observable object](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5806010167459840#The-Observable-object)
* [Subscribing to the observable](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5806010167459840#Subscribing-to-the-observable)

In the last example, we used the fromEvent() function to assist us in creating an observable. It makes an observable out of an element where it emits the event if the input event is triggered. We’re able to create an observable ourselves if we need more control over when an event is emitted.

**The Observable object**

We’ll be starting from scratch. A low-level observable can be created by creating a new instance of the Observable object. Let’s see what that looks like.

const { Observable } = rxjs;

const observable = new Observable(observer => {

    observer.next('Hello!');

    observer.complete();

    observer.error(new Error('An error has occurred.'));

});

In the example above, we’re grabbing the Observable object from rxjs. We’re creating a new instance out of it. We can pass in a function to the constructor function where we’ll be provided an object called observer. In some examples online, you may see this parameter aliased as subscriber.

The observer parameter is an object with functions for interacting with the observer. The functions can be used to emit a value, stop the observable completely, or throw an error. We have full control over the observable.

There are three functions we can use.

* next(): This will send a payload of data to the next observable.
* complete(): This notifies the observer that the observable is finished sending data.
* error(): This notifies the observer that an error has occurred in the observable.

Currently, we’re using the next() function to send a string as the payload. This will tell the observable to run the payload down the pipeline. Eventually, after the data has flowed through the pipeline, the observer will receive the data.

We can chain the pipe() function to add operators. Remember: operators will return a new observable. The next() function sends the payload down the pipeline. We’re not going to add pipes to this example. Pipes are entirely optional.

As for the complete() function, we don’t need to supply it anything.

The error() function can be provided with any value. It’s usually recommended to create a new instance of the Error object.

**Subscribing to the observable**

Let’s listen for the changes in the observable by creating an observer with the subscribe() function.

**OutputJavaScript**

const { Observable } = rxjs;

const observable = new Observable(observer => {

    observer.next('Hello!');

    observer.complete();

    observer.error(new Error('An error has occurred.'));

});

observable.subscribe({

    next(value) {

        console.log('Observable sent a value', value);

    },

    complete() {

        console.log('Complete');

    },

    error(err) {

        console.log(err.message);

    }

})

Observable sent a value

Hello!

Complete

In the example above, we’re calling the subscriber() function where we’ll wait for the observable to emit the next, complete, or error events.

If we switch to the Output panel, we’ll see the next() and complete() functions log their value. The error() function never runs because we’re calling the observer.complete() function *before* the observer.error() function. Once we call the observer.complete() function, the observable will not emit any data, including errors.

If we want to test if the error works, we’ll need to switch the calls in our callback function for the observable like so:

**OutputJavaScript**

const { Observable } = rxjs;

const observable = new Observable(observer => {

    observer.next('Hello!');

    observer.error(new Error('An error has occurred.'));

    observer.complete();

});

observable.subscribe({

    next(value) {

        console.log('Observable sent a value', value);

    },

    complete() {

        console.log('Complete');

    },

    error(err) {

        console.log(err.message);

    }

})

Observable sent a value

Hello!

An error has occurred.

This time, if we switch to the Output panel, we’ll see the error get logged but not the message from the observer’s complete() function. This is because of the same reason as before. If an error occurs, the observable is considered faulty. It will stop emitting data.

**Alternative Observer Syntax**

In this lesson, we'll learn about an alternative syntax for defining observers that you'll come across often.

One more thing before we jump into more advanced topics about RxJS: there’s an alternative syntax you’ll often encounter often whenever creating an observer. In fact, you’ve already seen the syntax when we created the cocktail application.

Instead of passing in an object to the subscribe() function, we can pass in functions. Here’s an example:

observable.subscribe(

····(value)·=>·console.log('Observable·sent·a·value',·value),·*//*·*next*

····(err)·=>·console.log(err.message),·*//*·*error*·

····()·=>·console.log('Complete'),·*//*·*complete*

);

In the example above, we’re directly passing in three functions. The first function will be used when an event is emitted. The second function is for when there’s an error. Lastly, the third function is for when the completion event is triggered.

The main reason developers use this syntax is that most of the time, they never need to define a function for when there’s an error or need to know when the observable’s completion event happens. Most of the time, developers just need to handle when an observable emits the next event.

**Understanding RxJS in Our Application**

Let’s use our newfound knowledge of RxJS to understand what's going on in our cocktails application.

**We'll cover the following**

* [Improving the request](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4581343830212608#Improving-the-request)

Let’s look at our cocktails application again. In the cocktail.service.ts, we’re using the http.get() method to request some data.

this.http.get('https://www.thecocktaildb.com/api/json/v1/1/search.php', {  
  params: {  
    s: query,  
  }  
});

The http.get() method will return an observable. It will emit the values retrieved by the API request. In the app.component.ts component class file, we’re using the observable by creating an observer with the subscribe() method.

this.cocktail.search(query).subscribe((response: any) => {  
  this.drinks = response.drinks;  
});

This is how observables are used in action. In our services, we can create observables that can emit data. We can create observers in our components where we’ll need the data.

**Improving the request**

We can improve how we create requests in our services using pipes and operators. We know that the response from the API will return an object with a property called drinks. This will represent an array of results based on the query.

Rather than returning the complete response, we’ll return the drinks arrays. This way, components don’t need to deal with anything else that might be returned.

In the cocktail.service.ts file, we’ll update the service to the following:

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

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

import { pluck } from 'rxjs/operators';

@Injectable({

  providedIn: 'root'

})

export class CocktailService {

  constructor(private http: HttpClient) { }

  public search(query: string) {

    return this.http.get('https://www.thecocktaildb.com/api/json/v1/1/search.php', {

      params: {

        s: query,

      }

    }).pipe(

      pluck('drinks')

    );

  }

}

We’re importing the pluck operator from the rxjs/operators package. Unlike before, we’re grabbing the function from a package. Angular internally uses RxJS. We’ll have access to it in our applications.

Since the http.get() method returns an observable; we’re more than free to chain the pipe() method to add some operators. We’re using the pluck() operator to grab the drinks property from the response. Since it’s an operator, it will return a new observable where it will emit the data.

We can update the app.component.ts class file to the following:

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

import { CocktailService } from './cocktail.service';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  drinks = [];

  query = '';

  constructor(private cocktail: CocktailService) { }

  search(query: string) {

    this.query = query;

    this.cocktail.search(query).subscribe((response: any) => {

      this.drinks = response;

    });

  }

}

In the callback function we passed into the subscribe() function, we’re setting the drinks property to the response. Previously, it was response.drinks. By using operators, we were able to simplify the code we need to write in components. They’re provided with only what’s necessary.