

---

# Introduction to RxJS

---

## Day 02

RxJS is a library for reacting to events and data. At the core is the Observable.

There are other types like Observers, Schedulers, Subjects and operators. All of these functions handle asynchronous events as collections of data.

ReactiveX is a combination of the Observer and the Iterator pattern. The library is based on functional programming which when applied to collections, provides an elegant way of managing sequences of events.

Events such as key presses, mouse movements, touch gestures, timer events, remote API calls, or single whole numbers can be thought of as a stream of data. It is all data to be consumed. Streams in this way can be considered an array that is always changing..

## Part 1 – Hot vs Cold Observables

In the simplest form, a *cold* observable will only become active once it detects a subscription. This is like placing a telephone call to a colleague or friend. You don't start talking until the other party answers. A *hot* observable will emit data regardless of who or what is listening. If no one subscribes, the data is just lost. This is like a radio broadcast, even if no one tunes in to that station, the DJ or host continues to talk.

Any code environment will work for this part, I will be using the environment we setup on Day01 Part01. This is a basic Node setup.

1. Enter this code to start the example of a cold observable:

```
//const { Observable } = rxjs;//if using a web based code environment
import { Observable } from 'rxjs';//for NodeJS environment
const rand$ = new Observable(
  obs => {
    obs.next( )
  }
);
```

In a cold observable two subscribers get their own (different) copies of values

2. The code above creates a new observable and is ready to emit a value. Lets emit a random between 0 and 100:

```
const { Observable } = rxjs;
const rand$ = new Observable(
  obs => {
    obs.next(Math.floor(Math.random() * (100)))
  }
);
```

3. Create an observer:

```
    obs.next(Math.floor(Math.random() * (100)))
  }
);
const obs1 = {
  next: aValue => console.log("we got " + aValue)
};
```

4. Now we simply subscribe. But notice that if we do not subscribe, nothing happens:

```
    obs.next(Math.floor(Math.random() * (100)))
  }
);
const obs1 = {
  next: aValue => console.log("we got " + aValue)
};
rand$.subscribe(obs1);
```

Just refresh the web page or run index.js again and you will see a different number each time.

5. So far, we have not really proven anything. Lets now create a second observer:

```
c);
const obs1 = {
  next: aValue => console.log("we got " + aValue)
};
const obs2 = {
  next: aValue => console.log("we got " + aValue)
};
rand$.subscribe(obs1);
```

6. Now subscribe using the second observer:

```
const obs2 = {
  next: aValue => console.log("we got " + aValue)
};
rand$.subscribe(obs1);
rand$.subscribe(obs2);
```

Just refresh the web page or run index.js again and you will see two different numbers each time. This proves that the observable is **cold**, it waits for a subscription then fires the `next()` method of the observer. The time of data generation is tightly coupled to the observable. Data is generated only when the `next()` method fires.

7. To make this observable **hot**, simply de-couple the data generation from the `next()` method of the observer:

```
const { Observable } = rxjs;
const rNumber = Math.floor(Math.random() * (100));
const rand$ = new Observable(
  obs => {
    obs.next(Math.floor(Math.random() * (100)))
```

Now, we can call `rNumber` from anywhere in the code. Before you had to subscribe to the observable in order for the `Math.floor()` to execute.

8. Now inside of the `next()` method of the observer, pass the `rNumber` value being generated by the `Math` object:

```
cconst rNumber = Math.floor(Math.random() * (100));  
const rand$ = new Observable(  
  obs => {  
    obs.next(rNumber)  
  }  
);
```

Now when you refresh, you get the same number for each subscription. Another way to make a cold observable hot, is by using the `share()` operator.



## Part 2 – ajax(),tap() and catchError()

One creation operator that is not used often is the `ajax()` creation operator. It is used mainly with APIs. The `tap()` operator is used for checking the status of data. It is often called a spy operator, it does not change the data in any way. The final operator in this section is the `catchError()` operator. This will handle errors and can be used anywhere, although it is used most times with API calls. As a bonus operator we will use `EMPTY` to return a null observable.

There are several public sites in the wild that provide testing data. For example I have used the *jsonplaceholder.typicode.com* site in other bootcamps. For this bootcamp we will use *catfact.ninja* which is a free API data provider that provides data on cats.

For this part, please unzip and have ready the *rxjs\_ts\_starter* code. Prior to beginning this part, please read and follow the instructions in the [README.md](#) file located in the root of *rxjs\_starter*.

1. Open the `index.ts` file located in the `src` folder and delete everything except the first line. Add/change to the following lines of code:

```
import { of } from 'rxjs';
import { ajax } from "rxjs/ajax";
import { catchError, map, tap } from "rxjs/operators";
```

2. The API we will be hitting is added to the code. I added a limit of 3 documents/records to the end point:

```
import { of } from 'rxjs';
import { ajax } from "rxjs/ajax";
import { catchError, map, tap } from "rxjs/operators";
const catAPI = "https://catfact.ninja/breeds?limit=3";
```

Note: you can change the limit or remove it, but it is easier to work with limited amounts of data when testing.

3. We now use the `ajax()` creation operator to generate an observable based on the data located from the API endpoint and subscribe to it:

```
const catAPI = "https://catfact.ninja/breeds?limit=3";
let catAPI$ = ajax(catAPI);
catAPI$.subscribe(result => console.log(result));
```

Note: you will need to run `npm run build` to incorporate any changes to the code.

4. We used the `ajax()` creation operator but that operator has a function `getJSON()` which will get us closer to the data:

```
const catAPI = "https://catfact.ninja/breeds?limit=3";
let catAPI$ = ajax.getJSON(catAPI);
catAPI$.subscribe(result => console.log(result));
```

After running this code, check the console window and drill down into the object, you will see the data there as an array of three objects.

5. In order to demonstrate the first operator `tap()`, let us `pipe()` our data before subscribing:

```
let catAPI$ = ajax.getJSON(catAPI);
catAPI$
  .pipe()
  .subscribe(result => console.log(result));
```

6. We can now insert our `tap()` operator into `pipe()` to view the data before it moves to another operator or vice-versa:

```
catAPI$
  .pipe(
    tap(value => console.log(value))
  )
  .subscribe(result => console.log(result));
```

You will notice that we get the same result twice, but `tap()` allows us to experiment with the data

7. Now that we have `tap()` we can try to drill into the response object that we got from `ajax()`. Lets add the data qualifier to the end of value:

```
catAPI$
  .pipe(
    tap(value => console.log(value.data))
  )
  .subscribe(result => console.log(result));
```

Although the IDE is reporting an error under `data`, we still get the three objects inside of the array. To remove the IDE error, just declare `value` as type `any`. Also you could comment the last log line, the one inside `subscribe()`.

8. Lets change the code a bit. Change *tap* to *map*, then add a new operator the `catchError()` operator.

```
catAPI$
  .pipe(
    map((value:any) => value.data),
    catchError(() => {})
  )
  .subscribe(
```

Note: `catchError()` needs a function that catches any errors thrown. Also it will show a syntax error until it detects a return from the `catchError()` function. Also the `map()` function returns a value instead of just logging it.

9. Continue to log the error:

```
catAPI$
  .pipe(
    map((value:any) => console.log(value.data)),
    catchError(err => {
      console.log("Error" + err);
    })
  )
  .subscribe(
```

10. The library has an operator called `EMPTY` and although it returns an observable, the observable behaves like the null operator, it does not return anything. Here is how it is used:

```
.pipe(
  map((value:any) => value.data),
  catchError(error => {
    console.log('error: ', error);
    return EMPTY;
  })
)
```

This operator is a creation operator so import it directly from rxjs.

11. Here is all the code for this section and the result in the console window of my firefox browser:

```

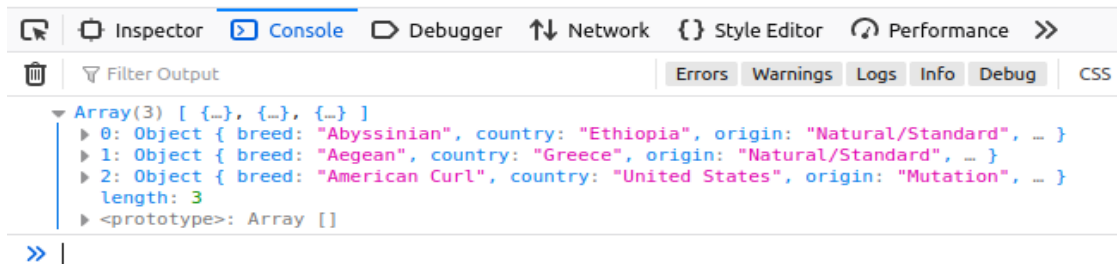
import { of, EMPTY } from 'rxjs';
import { ajax } from "rxjs/ajax";
import { catchError, map, tap } from "rxjs/operators";
const catAPI = "https://catfact.ninja/breeds?limit=3";
let catAPI$ = ajax.getJSON(catAPI);
catAPI$
  .pipe(
    map((value:any) => value.data),
    catchError(error => {
      console.log('error: ', error);
      return EMPTY;
    })
  )
  .subscribe(result => console.log(result));
)

```



## Just Testing

Use this for TS testing



To simulate an error, you could remove the 's' from the URL endpoint `/breeds`.





## Part 3 – Revisit to Subject plus Behavior Subject

For this part continue using the rxjs\_ts\_starter code. Mostly we will be changing the index.ts file.

On Day01 we looked at Subject and we said that it is both an Observable and an Observer at the same time. That example showed the Observable nature of Subject. In this part we look at the Observer side of the Subject and also introduce the Behavior Subject.

1. In this first example, this is the same code from Day01 Part 4. Just copy that code and add it to your index.ts file, build and run. However I have made a few changes below:

```
import { from, Subject, Observable } from 'rxjs';
const subject = new Subject();
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft');
```

Basically, I moved the first **next()** to the last line and commented out all other emissions. I also added the **from()** operator and **Observable** from the RxJS library.

2. To make the subject an observer, all we have to do is pass it to any Observable. First create an Observable object:

```
import { from, Subject, Observable } from 'rxjs';
const subject = new Subject();
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft');
const observable = new Observable();
```

3. Now pass the existing subject to the new Observable via the new Observable's `subscribe()` method and move the next line to the bottom:

```
import { from, Subject, Observable } from 'rxjs';
const subject = new Subject();
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the
data.`)
);
const observable = new Observable();
observable.subscribe(subject);
subject.next('Skillsoft');
```

4. However if we configure this new Observable to emit data, it would be picked up by the subscriber:

```
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the
data.`)
);
const observable = new Observable(o => {
  o.next("Hello")
});
observable.subscribe(subject);
subject.next('Skillsoft');
```

5. Also, remember we can create an Observable using the `from()` operator

```
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the
data.`)
);
const observable = from(["Tom"]);
observable.subscribe(subject);
subject.next('Skillsoft');
```

Remove everything to the right of the `=` sign and replace with the highlighted code.

6. Now if you move the `subject.next()` method above the new observable, you get both emissions:

```
import { from, Subject, Observable } from 'rxjs';
const subject = new Subject();
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft');
const observable = from(["Tom"]);
observable.subscribe(subject);
```

The original subject already has the `next()` method configured and `from()` made a second emission.

7. Lets now change Subject to **BehaviorSubject** and comment out the last two lines. The first thing that happens is that it asks for an initial value. Once you put that in, this operator works exactly like Subject. The only difference so far is that the initial value is emitted first:

```
import { BehaviorSubject } from 'rxjs';
const subject = new BehaviorSubject("BehaviorSubject");
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft');
```

8. A similar operator is the `AsyncSubject`. This operator will simply emit data from the last subscription:

```
import { from, AsyncSubject } from 'rxjs';
const subject = new AsyncSubject();
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
const s2 = subject.subscribe(
  (value:any) => console.log(`${value.toUpperCase()}, is the data.`)
);
subject.next('Skillsoft');
const observable = from(["Tom"]);
observable.subscribe(subject);
subject.next('Skillsoft2');
```

In this case Tom is the last emission, to the Observable

9. Another related operator is the **`ReplaySubject`**. This operator will emit data from the last `next()` emission. To see this better lets change our code to have three `next()` functions but just one subscription.

```
import { ReplaySubject } from 'rxjs';
const subject = new ReplaySubject();
subject.next('Skillsoft develop skills');
subject.next('Skillsoft has skills');
subject.next('Skillsoft rocks!');
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
```

When you run this code it behaves just like the regular Subject, so far.

10. What differentiates the `ReplaySubject` is the ability to pass a value into it's constructor, which will manage how many emissions to pass forward:

```
import { ReplaySubject } from 'rxjs';
const subject = new ReplaySubject(1);
subject.next('Skillsoft develop skills');
subject.next('Skillsoft has skills');
subject.next('Skillsoft rocks!');
const s1 = subject.subscribe(
  value => console.log(`The data is: ${value}.`)
);
```

When you run this code now, it emits just the last value emitted.

## Part 4 – User Input with RxJS

For this part there is a new starter file [d2p4\\_user\\_input\\_starter.zip](#). Actually it is the same set of files from the last part, only the .ts file has changed. It might be easier to just install this starter code than attempt to re-configure the last set of files.

Follow the instructions in the *readme* file. With this package I added a .css file just to make things spread apart on the browser. That file must be placed inside the *dist* folder once you have everything installed. You can find the .css file inside of the *src* folder once you unzip that zipped file. All of these operators have been imported already and are located on lines 1 and 2. You do not have to change the .html file for this section.

The files are based on TypeScript and HTML. Remember TypeScript gets compiled to plain JavaScript after WebPack does its bundling process. Usually though a developer would use React or Angular for situations like these.

1. This example requires a model, we need to describe the data we are collecting. Start with the [index.ts](#) file. Let's start with an interface for our data:

```
import { map, tap } from "rxjs/operators";
interface User {
  email?: string;
  password?: string;
}
let userModel: User = {};
```

This is a simple profile consisting of a user email and a password. The ? indicates to TS that this property may be null. The last line simply creates an empty user model instance.

2. The next order of business is to identify our form controls. This example does not use an actual form but the process is the same, identify each HTML element:

```
let userModel: User = {};
let buttonElement =
document.getElementById("submit") as HTMLButtonElement;
let emailInput =
document.getElementById("email") as HTMLInputElement;
let passwordInput =
document.getElementById("password") as HTMLInputElement;
let showOutput =
document.getElementById("show") as HTMLDivElement;
```

Here we use a slightly different approach to targeting HTML elements. Now the variable next to the **let** keyword, represents the HTML object associated with it. All of these HTML elements are already in the HTML file you unzipped.

3. The next six lines from points 3 and 4 could be condensed into just three but for clarity I am using separate lines:

```
let showOutput =  
    document.getElementById("show") as HTMLDivElement;  
let email$: Observable<InputEvent>;  
let password$: Observable<InputEvent>;  
let submit$: Observable<MouseEvent>;
```

Since we will react to three objects (controls) I create three observables of the event type.

4. Lets now create the actual Observables using the **fromEvent()** creation operator. We do this for each HTML object we are interested in:

```
let submit$: Observable<MouseEvent>;  
email$ = fromEvent<InputEvent>(emailInput, "input");  
password$ = fromEvent<InputEvent>(passwordInput, "input");  
submit$ = fromEvent<MouseEvent>(buttonElement, "click");
```

5. Now that we have wired up all our form objects and made them Observables, we can now subscribe to them in order to listen in. Lets start with the email field:

```
password$ = fromEvent<InputEvent>(passwordInput, "input");  
submit$ = fromEvent<MouseEvent>(buttonElement, "click");  
email$.subscribe(x=>console.log(x))
```

Remember to execute the command **npm run build**

If we just subscribed to this input event, then we get a lot of details we don't really need, see image below:

email:

password:

## Output here



If you want to see the code run, just do a build then open the index.html file in the browser using localhost.

- We need to get the value of what the user typed into that input box. One way is to *map* the input event to the event's target, then access its *value*:

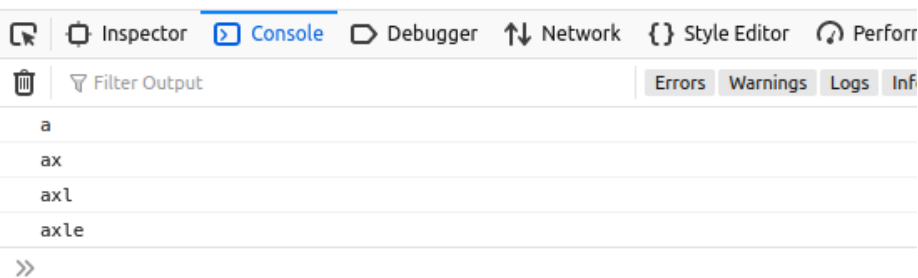
```
submit$ = fromEvent<MouseEvent>(buttonElement, "click");
email$
    .pipe(
        map((event:any) => event.target.value)
    )
    .subscribe(x=>console.log(x))
```

This gets us a little closer to the goal, but we will improve on this in the next point.

## Just Testing

email: password: 

## Output here



7. We already know that the email input field will raise an `InputEvent`, so let's describe our `map`'s parameter as such. In this way we don't have to use the **any** data type for this situation. Also we will convert `event.target` into an `HTMLInputElement` before accessing its value:

```
submit$ = fromEvent<MouseEvent>(buttonElement, "click");
//
email$
  .pipe(
    map(
      (event: InputEvent) =>
        (event.target as HTMLInputElement).value
    )
  ).subscribe(x=>console.log(x))
```

Note that typing `event.target` as an `HTMLInputElement` we do not get a TS error, but the code will work without this type conversion.



8. Now that we know that the code works so far, lets assign the value to our model instead of logging it:

```
email$
  .pipe(
    map(
      (event: InputEvent) => (event.target as
HTMLInputElement).value
    )
  ).subscribe(email => userModel.email = email)
```

9. Now that email works, we could apply the same technique to the *password* field. Remember you can insert the `tap()` operator anywhere inside of the `pipe()` operator to see your values. Just duplicate the code for the *password* field:

```
password$
  .pipe(
    map((event: InputEvent) =>
      (event.target as HTMLInputElement).value)
  )
  .subscribe(password => userModel.password = password)
```

10. If we got this far, it means that we have the user's profile and can now store that data in the model and use it to challenge an API login endpoint:

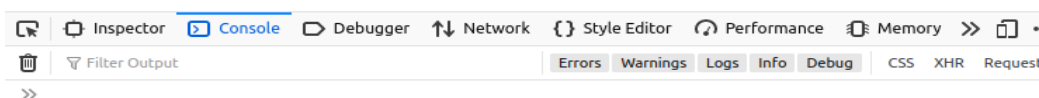
```
submit$
  .subscribe(
    () => showOutput.innerHTML = JSON.stringify(userModel)
  );
```

## Just Testing

email:

password:

```
{"email":"axle.barr@skillsoft.com","password":"1234"}
```



## Part 5 – Flattening Operators

Flattening operators perhaps should be called higher order mapping operators. There really is not much flattening going on, these are more combination operators than flattening.

In higher-order mapping, values from the source Observable are mapped into an inner Observable. The result is a higher-order Observable. We can treat the result of this union as any other Observable. Basically one Observable consumes another.

For Part5 you could continue to use the code from Part4.

1. This example should look very familiar now. The Observable is created using the `of()` operator and is referenced by `source$`. Once we subscribe to `source$` we can then log the values of 1000 and 2000

```
import { of } from 'rxjs';
import { concatMap } from 'rxjs/operators';
const source$ = of(1000, 2000);
source$.subscribe(x => console.log(x));
```

2. Using the same code, comment out the subscription. Use the `pipe()` operator and pass into it the `concatMap()` operator. Take the value from our original Observable and pass it to another `of()` operator generating the words *Skill* and *Soft*:

```
const source$ = of(1000, 2000);
//source$.subscribe(x => console.log(x));
source$
  .pipe(
    concatMap(val => of("Skill", "Soft"))
  ).subscribe(x=>console.log(x))
```

You will see the values Skill and Soft being printed out. The value in *val* is lost. In fact for this situation you can eliminate it and just use an empty pair of parenthesis:

**`concatMap(() => of("Skill", "Soft"))`**

3. Of course we could involve `val` if we did something like this:

```
source$
  .pipe(
    concatMap(val => of(`Skillsoft ${val}`))
  ).subscribe(x=>console.log(x))
```

In this case we should see two outputs, one for 1000 and another for 2000

4. Lets now see how we can take advantage of this feature. Change the HTML file from the previous part to the code below:

```
<h1>Just Testing</h1>
<div>
  <label for="searchWord">query word:</label>
  <input type="searchWord" id="searchWord" size="20"
  maxlength="24" />
</div>
<div>
  <button id="submit" type="button">Submit</button>
</div>
```

Simply remove the *password* field and change the *email* field to something like *searchWord*. This is all the code between the `<body>` tags. Remember to do this in the `src` folder.

5. We will be using a few more operators for this part, so import as necessary until your imports look like this one below:

```
import { fromEvent, Observable, of } from "rxjs";
import { map, concatMap, tap, catchError } from
  "rxjs/operators";
import { ajax } from "rxjs/ajax";
```

6. On the TS code side, the button will remain the same as in Part 4, but remove all the code that refers to *password*, *show* and the *model*:

```
let buttonElement =
  document.getElementById("submit") as HTMLButtonElement;
let emailInput =
  document.getElementById("email") as HTMLInputElement;
let submit$: Observable<MouseEvent>;
submit$ = fromEvent<MouseEvent>(buttonElement, "click");
```

We also do not need the `email$ Observable`

7. Remove all the code below the `submit$` line. Replace all that code with the `submit$` pipe and subscribe:

```
submit$ = fromEvent<MouseEvent>(buttonElement, "click");
submit$
.pipe()
.subscribe()
```

At this point, you should have three imports and three lets then the code you see above.

8. Change the word email to `searchWord` or something similar:

```
let buttonElement =
document.getElementById("submit") as HTMLButtonElement;
let searchWordInput =
document.getElementById("searchWord") as HTMLInputElement;
let submit$: Observable<MouseEvent>;
```

The name you choose should match the field on the HTML document.

9. Access the value the user types into the `searchWord` input field, then `tap()` the value to make sure it is coming through ok:

```
submit$
.pipe(
  map(() => searchWordInput.value),
  tap(val => console.log(val))
)
.subscribe()
```

You may remove the `tap()` or leave it

10. Once we get a value from that `search` field, we can now apply the `concatMap()` operator to pass that value into a different Observable:

```
submit$
.pipe(
  map(() => searchWordInput.value),
  tap(val => console.log(val)),
  concatMap(val => ajax(val))
)
```

Notice `ajax()`, we used this before

11. All we have to do now is pass in an API endpoint and concatenate the value we got from the search box:

```
tap(val => console.log(val)),
concatMap(
  val =>
    ajax(`https://jsonplaceholder.typicode.com/${val}`)
)
)
```

Now, complete the `subscribe()` function and add on the `response` property to the returned object

12. Test with the search word of *posts*:

## Just Testing

Search Word:



Note: there is a bug in my system so I get a duplicated output. Also the posts that appear above the first array is from the `tap()` operator. We will solve some of these issues in the next part.

13. Here is the entire index.ts file, cleaned up a bit:

```
import { fromEvent, Observable } from "rxjs";
import { map, concatMap } from "rxjs/operators";
import { ajax } from "rxjs/ajax";
//
let buttonElement =
document.getElementById("submit") as HTMLButtonElement;
let searchWordInput =
document.getElementById("searchWord") as HTMLInputElement;
let submit$: Observable<MouseEvent>;
submit$ = fromEvent<MouseEvent>(buttonElement, "click");
//
submit$ = fromEvent<MouseEvent>(buttonElement, "click");
submit$
    .pipe(
        map(() => searchWordInput.value),
        concatMap(val =>
ajax(`https://jsonplaceholder.typicode.com/${val}`))
    )
    .subscribe(
        posts => console.log(posts.response)
    );
```

## Part 6 – More on Flattening Operators

In Part 5 we used the `concatMap()` operator as a higher order Observable. This creates a few more issues we as developers have to deal with. In this part we extend the code from Part 5 to deal with some of these issues.

1. Unless we hard code search terms into something like a Drop Down List control, the user may enter incorrect search terms. Lets introduce the `catchError()` operator to handle this situation. However, since `ajax()` returns an Observable, we can apply the `pipe()` operator here, inside `concatMap()` :

```
tap(val => console.log(val)),
concatMap(
  val => ajax(`https://jsonplaceholder.typicode.com/${val}`)
    .pipe()
)
)
```

2. Now just pass the `catchError()` operator into the `pipe()` stream we just inserted:

```
concatMap(
  val => ajax(`https://jsonplaceholder.typicode.com/${val}`)
    .pipe(
      catchError()
    )
)
```

You will need to import `catchError()`

3. The `catchError()` method takes an function that has access to the error object if one was generated. Configure the method with an appropriate function:

```
concatMap(val =>
  ajax(`https://jsonplaceholder.typicode.com/${val}`)
    .pipe(
      catchError(error => {throw new Error("API Error " + error)}),
    )
)
```

4. At this point, we will not see any errors as the error object is not being captured. It is actually being passed into the `subscribe()` function. Remember an Observable has three potential exit points, *next*, *error* and *complete*. Lets use this information in the *subscribe* function. First, in the `subscribe()` function wrap the current code into curly braces:

```
.subscribe(  
  { posts => console.log(posts.response) }  
)
```

Note: for this part of the code to work, remove the `.responses` from `posts`, we will put it back in later.

5. Add the first of the three possible *subscribe* function handlers, *next*:

```
.subscribe(  
  {next : (posts:any) => console.log(posts)}  
)
```

Also, wrap the parameter in parenthesis. Then, define that parameter as *any*, this would remove the IDE complaining about not knowing what kind of data it is handling.

6. Now we can complete the other functions:

```
.subscribe(  
  {  
    next : (posts:any) => console.log(posts) ,  
    error: err => console.log('Error:', err) ,  
    complete: () => console.log('Done!')  
  }  
)
```

Remember to test for an error in this part of the code, change the url to something that will not be found.



7. Here is the all the code from submit\$:

```
submit$
  .pipe(
    map(() => searchWordInput.value),
    concatMap(val =>
      ajax(`https://jsonplaceholder.typicode.co/${val}`)
        .pipe(
          catchError(error => {throw new Error("API Error "+error)}),
        )
    )
  )
  .subscribe(
    {
      next : (posts:any) => console.log(posts.response),
      error : err => console.log(err),
      complete : () => console.log("done")
    },
  )
```

Error on  
purpose

Remember to test for an error in this part of the code, change the url to something that will not be found

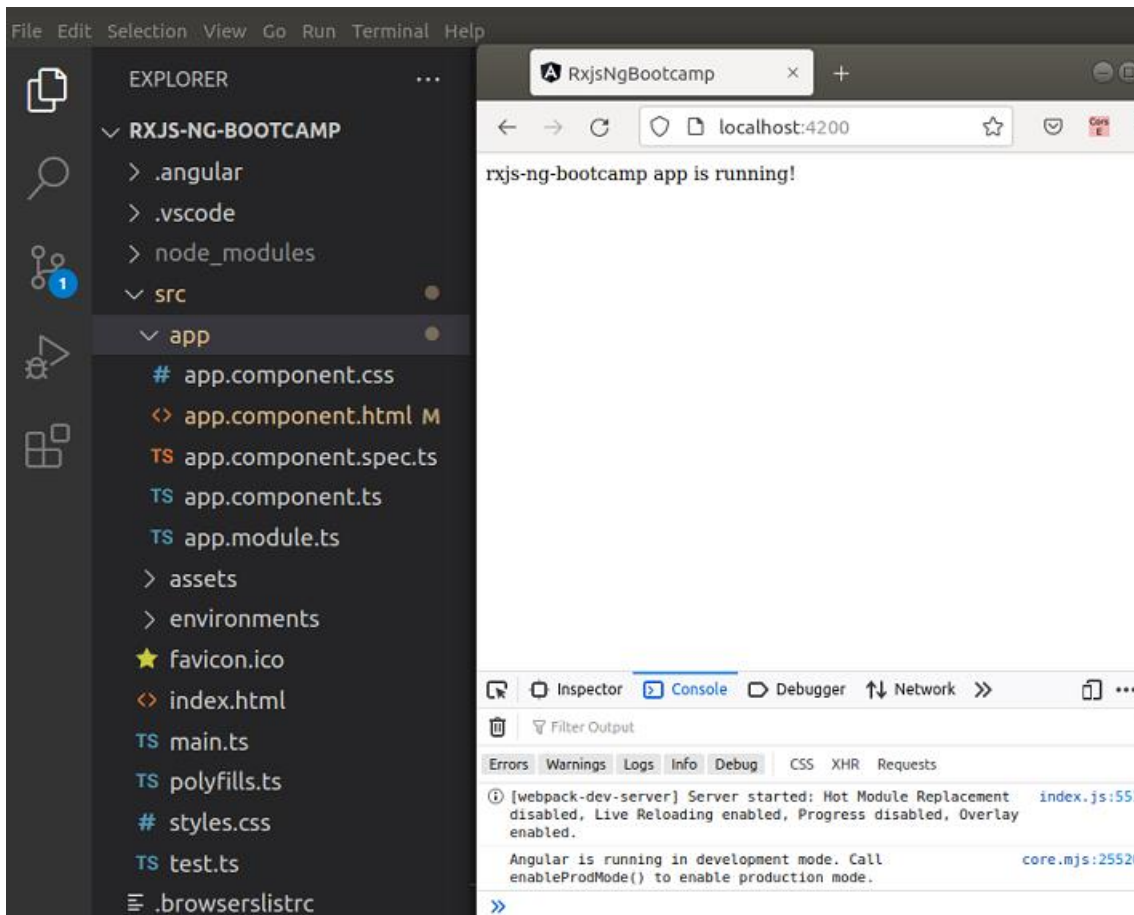
*fnd of section*

## Part 7 – Angular App Setup

We shift now to building an Angular application, just a boilerplate but it has several learning points in the process.

Angular uses the RxJS library heavily. In fact just installing the CLI will install the library by default. We will install an Angular app and show the difference between an imperative approach and a reactive one. At the same time we will use a new operator. Appendix C has instructions on how to build an Angular app from just the CLI commands.

1. If you are using the provided starter file, unzip it in a folder and run the Node command `npm install` to install the basic boiler-plate application. If you then open a terminal window from VS Code and run `ng serve`, you can then open a browser and navigate to the 4200 port on *localhost* to see the app there.



The image above shows the rxjs-ng-bootcamp app in VS Code as well as what the app looks like in a browser window (Firefox).

2. The first change we make is to import the *HttpClientModule* from *@angular/common*. Open the `app.module.ts` file and add the following line along with the other imports:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
```

3. Close the `app.module.ts` file, we will not be using it anymore. Now open the `app.component.ts` file and import the *HttpClient* and *Observable* modules:

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/internal/Observable';
//
@Component({
```

Remember to add a comma above the inserted module, this is an array.

4. Lets now use the *HttpClient* to make a *get* request to a public API setup for testing purposes. First we need to inject this service into the component via the component's constructor:

```
})
export class AppComponent {
  constructor(private http: HttpClient) {
  }
  title = 'rxjs-ng-bootcamp';
}
```

5. At this point we don't know the structure of our data so we declare a variable as type *any*. Also we use then `ngOnInit()` method to make our call:

```
constructor(private http: HttpClient) {
}
title = 'rxjs-ng-bootcamp';
users : any;
//
ngOnInit() {
}
}
```

Usually if you use the `ng generate component` command, you will get some of this code pre-inserted.

## 6. Add code to get the data using the httpClient:

```

users : any;
//
ngOnInit() {
  this.http.get<any>("https://jsonplaceholder.typicode.com/users")
    .subscribe();
}

```

Remember an Observable will not reveal it's secrets unless we subscribe to it

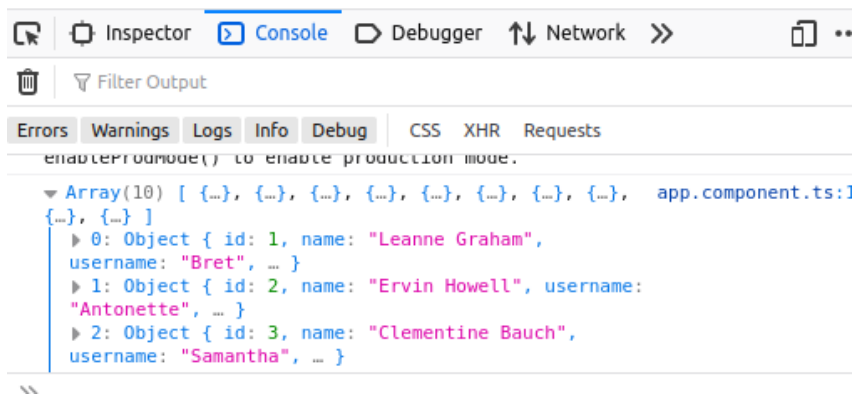
## 7. To see the response, simply provide a bucket, then log that bucket:

```

ngOnInit() {
  this.http.get<any>("https://jsonplaceholder.typicode.com/users")
    .subscribe(
      response => console.log(response)
    );
}

```

## 8. If all goes well, you should see something like the image below, in your developer's console window:



Notice that the response is already an array. This is the jsonplaceholder website, nothing to do with our code.



*End of section*



## Part 8 – Reactive Version of Angular App

The app in Part 7 has several problems. The code could actually lead to callback hell. The component is strongly coupled with the data. That makes the component more difficult to test. Data should be de-coupled from the component, so that other components could use it and don't need to worry about changes to that data. Mutation should be done locally. In order to make this app more reactive, we need to port the data access part to an angular *service*. We can then import that service anywhere it is needed.

A note on the zipped file (apidemo\_d2\_starter\_p7.zip), if you chose to use it. You will need to unzip that file, rename the folder to apidemo and run `npm install` in order to bring it to server level.

1. From a terminal window run the command `ng g s getUsers`. Make sure you are in the folder where the app is located. If successful, you will see two files created. The `.ts` file is the one we are interested in. Notice that the class is **GetUsersService**, all uppercase starting words. Also notice the the file name has changed slightly compared to what we typed. (`ng g s getUsers --skip-tests`).
2. Import into the `get-users.service.ts` file, the `Http` service and the `Observable` module that comes with angular into this new service. So exactly like we did before inside of the component.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/internal/Observable';
//
@Injectable({
```

Note, you do not have to do anything with `ap.module.ts`

3. Inject the `http` service just like before:

```
export class GetUsersService {
  constructor(private http:HttpClient) { }
}
```

Remember to add a comma above the inserted module, this is an array.

4. Lets now use the Observable module we imported to create a new Observable property to store the data we will be getting:

```
export class GetUsersService {
  users$: Observable<any>;
  constructor(private http:HttpClient) { }
}
```

For now, we state that users\$ is an observable and we do not as yet know the structure of the data. The bang (!) after user\$ is to prevent a TS error.

5. At this point we can begin to request the data and store it in our Observable. We need to put our request code into a function like `getAllUsers()`

```
getAllUsers() : Observable<any>{
  this.users$ =
    this.http.get<any>(
      "https://jsonplaceholder.typicode.com/users"
    );
  return this.users$;
}
```

6. Remember we are returning users\$ and it was defined as an Observable. Remember an Observable will not reveal it's secrets unless we *subscribe* to it, which we do in the component, NOT in the service. Also remember the data is already in array form, so not much to do with it at this point.
7. Back in the component, so `app.component.ts`, remove anything that has to do with Observables or the HttpClient. Add a property to accept the Observable being returned from the service:

```
export class AppComponent {
  constructor() {
  }
  title = 'rxjs-ng-bootcamp';
  users$: Observable<any>;
  //
  ngOnInit() {
  }
}
```

Remove the two imports to Observable and HttpClient

8. Now import the *GetUsersService* service and inject it into the constructor. Also change users to an Observable type receiving **any** type of data. Also we can use the `ngOnInit()` method to accept the Observable:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/internal/Observable';
import { GetUsersService } from "../get-users.service";
//
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private getUsers:GetUsersService) {
  }
  title = 'rxjs-ng-bootcamp';
  users$: Observable<any>;
  //
  ngOnInit() {
    this.users$ = this.getUsers.getAllUsers();
  }
}
```

9. If you want you could subscribe to the data returned from the service to verify that it is the same data we saw earlier:

```
ngOnInit() {
  this.users$ = <any>this.getUsers.getAllUsers();
  this.users$.subscribe(
    data => console.log(data)
  )
}
```

You may need to add the type of data being returned, `<any>`. Uncomment this subscription once you have verified the data.

10. Once you confirm that you have an array we can now transfer the work of subscribing to the data to the front end (HTML). Open the `app.component.html` file and add the following lines:

```
<div *ngFor="let user of (users$ | async)">
  <div>{{user.name}} uses the name: {{user.username}}</div>
</div>
```

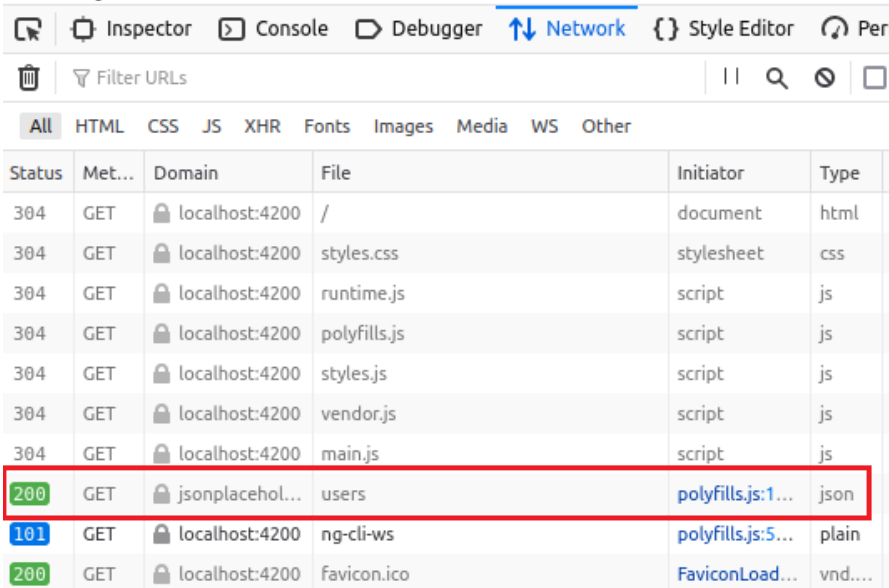
Notice that we pipe the observable to async function that does all of the subscribing and unsubscribing for us. Notice that the response is already an array. This is the jsonplaceholder website, nothing to do with our code.

## Part 9 – Multiple API Calls

The code seems to work well but there is an issue. If we had an api where we needed to get the same data but in different components, we will be making multiple HTTP requests. Lets see how the RxJS library can help in this situation.

1. At the end of point #9 in Part08 above I asked you to uncomment the subscription we used to log the data. Please do it now if you have not done it already.
2. Clear the console window and refresh the browser. Open the Network tab (in the console window area of the browser). Try to locate the call to localhost:4200 (where it found the users file). This is how it looks like in Mozilla Firefox, but it should be similar on all browsers:

Ervin Howell uses the name: Antonette  
Clementine Bauch uses the name: Samantha  
Patricia Lebsack uses the name: Karianne  
Chelsey Dietrich uses the name: Kamren



Status	Met...	Domain	File	Initiator	Type
304	GET	localhost:4200	/	document	html
304	GET	localhost:4200	styles.css	stylesheet	css
304	GET	localhost:4200	runtime.js	script	js
304	GET	localhost:4200	polyfills.js	script	js
304	GET	localhost:4200	styles.js	script	js
304	GET	localhost:4200	vendor.js	script	js
304	GET	localhost:4200	main.js	script	js
200	GET	jsonplacehol...	users	polyfills.js:1...	json
101	GET	localhost:4200	ng-cli-ws	polyfills.js:5...	plain
200	GET	localhost:4200	favicon.ico	FaviconLoad...	vnd...



- Now uncomment the code I asked you to comment from point #1 above and refresh the browser. You should now see two network requests for that users file.
- Now over in the template, add a new pair of `<div>` tags to get the zipcode of these users:

```
<div *ngFor="let user of (users$ | async)">
  <div>{{user.name}} uses the name: {{user.username}}</div>
</div>
<div *ngFor="let user of (users$ | async)">
  <div>{{user.address.zipcode}}</div>
</div>
```

- If you check the network requests there should now be three for the users file:

Ervin Howell uses the name: Antonette  
 Clementine Bauch uses the name: Samantha  
 Patricia Lebsack uses the name: Karianne  
 Chelsey Dietrich uses the name: Kamren

The screenshot shows the Chrome DevTools Network tab with the 'Network' panel selected. The 'Filter URLs' field is empty. The 'All' tab is selected, showing a list of network requests. The 'users' file is requested three times, each with a status of 200 and a type of json. The requests are highlighted with a red box.

Status	Met...	Domain	File	Initiator	Type	Transferred
304	GET	localhost:4200	/	document	html	cached
304	GET	localhost:4200	styles.css	stylesheet	css	cached
200	GET	localhost:4200	runtime.js	script	js	6.82 KB
304	GET	localhost:4200	polyfills.js	script	js	cached
304	GET	localhost:4200	styles.js	script	js	cached
304	GET	localhost:4200	vendor.js	script	js	cached
200	GET	localhost:4200	main.js	script	js	10.16 KB
101	GET	localhost:4200	ng-cli-ws	polyfills.js:5...	plain	129 B
200	GET	jsonplaceholder...	users	polyfills.js:1...	json	cached
200	GET	jsonplaceholder...	users	polyfills.js:1...	json	cached
200	GET	jsonplaceholder...	users	polyfills.js:1...	json	cached
200	GET	localhost:4200	favicon.ico	FaviconLoad...	vnd....	cached

- In the service file, so `get-users.service.ts` file, import the `shareReplay()` operators:

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/internal/Observable';
import { shareReplay } from 'rxjs/internal/operators/shareReplay';
```

7. At the first point of contact with the API, we need to apply the `shareReplay()` operator. So in this case it is in the `getAllUsers()` function of the `get-users.service.ts` file:

```
constructor(private http:HttpClient) { }
getAllUsers() : Observable<any>{
  this.users$ =
    this.http.get<any>(
      "https://jsonplaceholder.typicode.com/users"
    ).pipe(shareReplay());
  return this.users$;
}
```

8. Perform the same check with this new code in place. You should now only see one network request for that users file.

9. (Optional) if you want to see the users and their telephone numbers use this style:

<pre>.float-container {   border: 3px solid #fff;   padding: 20px; }  .float-child {   width: 500px;   float: left;   padding: 20px;   border: 2px solid red; }</pre>	<pre>&lt;div class="float-container"&gt;   &lt;div class="float-child"&gt;     &lt;div *ngFor="let user of (users\$  async)" &gt;       &lt;div&gt;&lt;strong&gt;{{user.name}} &lt;/strong&gt;uses the name: {{user.username}}&lt;/div&gt;     &lt;/div&gt;   &lt;/div&gt;   &lt;div class="float-child"&gt;     &lt;div *ngFor="let user of (users\$  async)"&gt;       &lt;div&gt;{{user.address.zipcode}}&lt;/div&gt;     &lt;/div&gt;   &lt;/div&gt; &lt;/div&gt;</pre>
---	---

## Appendix A – Use Case for Hot Observables

In this section we take a look at creation Observables, classes from the library that create Observables automatically. There are several of them but `from` and `of` are the most popular.

A hot observable will emit data even if no one is listening. Here the user may move the mouse and click on any part of the document. In this way, the observable `down$` will generate data whenever the user performs the mouse-down click. If we wanted to use that data, just subscribe to it:

```
const { fromEvent } = rxjs;
const down$ = fromEvent(document, 'mousedown');
down$.subscribe(
  (e) => console.log(e.clientX)
);
down$.subscribe(
  (e) => console.log(e.clientY)
);
```

## Appendix B – TS Environment Setup Linux

In this section we will setup a basic folder structure and config files for a very simple TS, Webpack, Node application. Note the [rxjs ts starter.zip](#) file is already built and available.

1. Create a folder, mine is [rxjs\\_starter](#).
2. Run the command to build a basic package.json file:  
`npm init -y`
3. Install the following packages:  
`npm install rxjs webpack typescript ts-loader`
4. Install the following packages as development dependencies:  
`npm install --save-dev html-webpack-plugin webpack-cli`

5. Open the folder in VS Code. In the `package.json` file, delete the option under scripts and add this instead:

```
"build": "webpack"
```

6. Create a `webpack.config.js` file, then add the following code:

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  devtool: 'eval-source-map',
  entry: './src/index.ts',
  mode: 'development',
  module: {
    rules: [
      {
        test: /\.ts$/,
        use: 'ts-loader',
        include: [path.resolve(__dirname, 'src')]
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: 'src/index.html'
    })
  ],
  resolve: {
    extensions: ['.ts', '.js']
  },
  output: {
    publicPath: './',
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

7. Create a `tsconfig.json` file and add this content to it:

```

{
  "compilerOptions": {
    "outDir": "public",
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "esnext",
    "target": "es2016",
    "allowJs": true,
    "moduleResolution": "node"
  }
}

```

8. Add a `src` folder inside the original `rxjs_starter` folder then add this `index.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>Reactive Extensions</title>
    <link rel="icon" href="data:,">
  </head>
  <body>
    <main>
      <h1>Just Testing</h1>
      <div>Use this for TS testing</div>
    </main>
  </body>
</html>

```

This is the same file from Day01 but adjusted for TS

9. In the same `src` folder, add an empty ts file called `index.ts`. Alternatively you can add some content to see if it all works:

```
import { Observable } from 'rxjs';
//
const myObs$ = new Observable<string>(obs => {
  obs.next('Skillsoft');
  obs.next('Rocks');
});
//
myObs$.subscribe(
  value => console.log(value)
);
```

For this TS option, the RxJS library would have been downloaded using the [package.json](#) file. It is therefore available and you can import the various classes.

For testing this setup environment, first run the command `npm run build`. If all goes well you should see an additional folder called `dist`. Inside of that folder there should be two files, right click on the `index.html` file and open with *Live Server*. Note Live Server is part of the VS Code environment. Any change made to `index.html` in the `src` folder is reflected in the `index.html` file in the `dist` folder, after running the *build* command.



## Appendix C – Installing Angular

Angular uses the RxJS library and is an excellent technology to practice using the various operators. We will install Angular, then configure it to work with some of the examples in Day 02 of the bootcamp.

1. If you do not currently have the Angular CLI installed, run this code from the command window to install it:

```
npm install -g @angular/cli
```

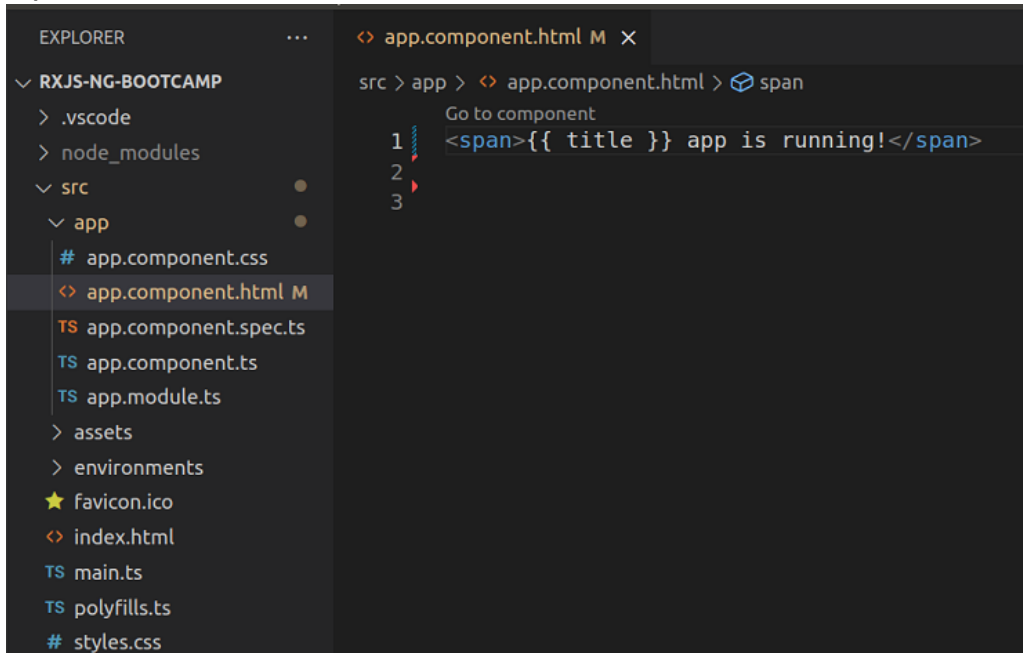
The -g flag is the global install flag

2. Once the install is done, this code will install a new application named rxjs-ng-bootcamp:

```
ng new rxjs-ng-bootcamp
```

Run this code from a folder and answer N for routing. Accept the normal CSS when asked.

3. Open the folder in VS Code and most of our work will be in the src/app folder:



Remove all the code from the `app.component.html` file except for the line shown in the image. This is just to see something on the browser window.

4. Open a Terminal window from VS Code (or from the normal terminal window) and run the following command to build/serve the app:

```
ng serve
```

This command will serve the app at port 4200 on your localhost server. Open your browser to that port and you should see the words: `rxjs-ng-bootcamp app is running!`

5. If you are using the zipped file provided, just copy that file into your favorite folder and unzip it. You should see a folder called rxjs-ng-bootcamp.
6. This folder now needs to be built with Node, not Angular. From a terminal window (or VS Code) type the command below to install and build the application:

```
npm install
```

7. If the install goes well, repeat #4 above and the application is now ready for the bootcamp. Note if you add the `-o` flag after the command in #4, the browser will open automatically to the correct path.

## Appendix D – More on Flattening Operators

In Part 5 we used the `concatMap()` operator as a higher order Observable. This creates a few more issues we as developers have to deal with. In this part we extend the code from Part 5 to deal with some of these issues.

1. Unless we hard code search terms into something like a Drop Down List control, the user may enter incorrect search terms. Lets introduce the `catchError()` operator to handle this situation. However, since `ajax()` returns an Observable, we can apply the `pipe()` operator here, inside `concatMap()` :

```
tap(val => console.log(val)),
concatMap(
  val => ajax(`https://jsonplaceholder.typicode.com/${val}`)
    .pipe()
)
)
```

2. Now just pass the `catchError()` operator into the `pipe()` stream we just inserted:

```
concatMap(
  val => ajax(`https://jsonplaceholder.typicode.com/${val}`)
    .pipe(
      catchError(error => of(`API Server error\n${error}`))
    )
)
```

You will need to import `catchError()` and `of()` . The `\n` will create a new line.

3. At this point, we will not see any errors as the error object is not being captured. It is actually being passed into the `subscribe()` function. Remember an Observable has three potential exit points, *next*, *error* and *complete*. Lets use this information in the `subscribe` function. First, in the `subscribe()` function wrap the current code into curly braces:



```
.subscribe(
  { posts => console.log(posts.response) }
)
```

Note: for this part of the code to work, remove the `.responses` from `posts`, we will put it back in later.

4. Add the first of the three possible *subscribe* function handlers, `next`:

```
.subscribe(
  { next : (posts:any) => console.log(posts) }
)
```

Also, wrap the parameter in parenthesis. Then, define that parameter as *any*, this would remove the IDE complaining about not knowing what kind of data it is handling.

5. Now we can complete the other functions:

```
.subscribe(
  {
    next : (posts:any) => console.log(posts) ,
    error: err => console.log('Error:', err) ,
    complete: () => console.log('Done!')
  }
)
```

Remember to test for an error in this part of the code, change the url to something that will not be found.

6. Here is the all the code from submit\$:

```

submit$
  .pipe(
    map(()=>searchWordInput.value),
    concatMap(
      val => ajax(`https://jsonplaceholder.typicode.co/${val}`)
        .pipe(
          catchError(error => {
            return of(`API Server error\n${error}`)
          })
        )
    )
  )
  .subscribe(
    {
      next : (posts:any) => console.log(posts),
      error: err => console.log('Error:', err),
      complete: () => console.log('Done!')
    }
  );

```

Remember to test for an error in this part of the code, change the url to something that will not be found

7.

8. The above code does show us the error, but since we forwarded that error via the `of()` operator, it comes through as data. Below we will improve on the error handling.

9. Instead we could re-throw the error, change the code to what you see below:

```

submit$
  .pipe(
    map(()=>searchWordInput.value),
    concatMap(
      val => ajax(`https://jsonplaceholder.typicode.com/${val}`)
        .pipe(
          catchError(err => {
            console.log('API Error ' + error);
            return throwError( ()=> err );
          })
        )
    )
  )
  .subscribe(

```

Here we log the initial error, then re throw a new error to be handled by the error handler in the `subscribe()` method.

10. Now we can replace the `.response` to the posts in `subscribe()`:

```
.subscribe(  
  {  
    next : (posts:any) => console.log(posts.response) ,  
    error: err => console.log('Error:', err) ,  
    complete: () => console.log('Done!')  
  }  
)
```

This is the recommended way to handle *ajax* errors according to the documentation. So now if an error occurred, we handle it otherwise we send it to the error handler in the *subscribe* method.



*End of section*



## Appendix E – Hot/Cold Observables Example

The previous example of hot/cold Observables showed how two subscriptions will get the same data with a hot Observable. In this example we make a cold Observable hot but subscribe to the hot Observable at different times.

1. Here is a cold observable, all subscriptions get the same data:

```
import { of } from 'rxjs';
const source$ = of(1,2,3);
source$.subscribe(
  data => console.log(data)
);
//
source$.subscribe(
  data => console.log(data)
);
//
source$.subscribe(
  data => console.log(data)
);
```

2. Here is a cold observable, but the subscriptions are at different times:

```
import { interval, take } from 'rxjs';
let source$ = interval(500)
.pipe(
  take(5)
);
source$.subscribe(
  data => console.log("First Subscription: " + data)
);
setTimeout(() => {
  source$.subscribe(
    data => console.log("Second Subscription: " + data)
  );
}, 2000);
```

In this example, even though the second subscription joins the party two seconds later, it still gets the same data.

3. Here we employ the `share()` operator. This operator returns a new Observable that shares the original Observable. Because the Observable is multicasting it makes the stream hot. Each subscription gets the same data but depending on the time of subscription may not get all the data

```
import { interval, take, share } from 'rxjs';
let source$ = interval(500)
  .pipe(
    take(5),
    share()
  );
source$.subscribe(
  data => console.log("First Subscription: " + data)
);
setTimeout(() => {
  source$.subscribe(
    data => console.log("Second Subscription: " + data)
  );
}, 2000) => console.log(data)
);
```