Transforming Responses Using the Map Operator in Angular

## **Introduction**

In this guide, we’re going to learn about the map operator. The map operator, basically, helps us to transform data using an observer. A map operator is a function that builds on the observable’s foundation to enable sophisticated manipulation of collection.

## **Syntax of Map Operator**

The map operator applies a given project function to each value emitted by the source Observable and emits the resulting values as an observable.

### **Syntax**

map<T, R>(project: (value: T, index: number) => R, thisArg?: any): OperatorFunction<T, R>

The “project” is a function to apply to each value emitted by the source observable. The index parameter is the number “i” for the “i”-th emission that has happened since the subscription, starting from the number 0. The “thisArg” is an optional argument to define what the index parameter is in the project function. The default value is undefined. The map operator returns an observable, OperatorFunction<T, R>, that emits the values from the source observable transformed by the given project function. Similar to the well-known Array.prototype.map function, this operator applies a projection to each value and emits that projection in the output observable. Let’s look at an example below where we are using a map operator. On every click on the view, we are capturing the position clientX event and logging it using the console log.

1

2

3

4

5

6

7

import { fromEvent } from 'rxjs';

import { map } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');

const positions = clicks.pipe(map(ev => ev.clientX));

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

```

typescript

## **Using Map Operator in Angular**

The map operator takes configuration options, and they return a function that takes a source observable. When executing the return function, the operator observes the source observables emitted values, transforms those values, and returns a new observable of those transformed values. Let's take a look at an example below.

1

2

3

4

5

6

7

8

9

10

11

12

13

import {map} from 'raj’s/operators';

const nums = of (1, 2, 3);

const mulValues = map ((val: number) => val \* 2);

const mulNums = mulValues (nums);

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

// Outputs

// 1

// 4

// 6

typescript

In the above example, we are importing the map operator from the operator’s library of RxJS. By using the map operator, we are transforming values to values multiplied by two.

## **Using the Map Operator with the Pipe Function**

We can use pipes to link operators together. Pipes let you combine multiple functions into a single function. The pipe() function takes the functions you want to combine as it's arguments and returns a new function that, when executed, runs the composed functions in sequence. A set of operators applied to an observable is a recipe. A recipe is a set of instructions for producing the values you are interested in. By itself, the recipe doesn't do anything. You need to call subscribe() to produce a result through the recipe.

In the below example, let’s look at how both the filter and map operators are used to transform the data using the pipe. First, we will check if the values are even and then we’ll multiply those values by two.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

import {filter, map} from 'raj’s/operators';

const nums = of (1, 2, 3, 4, 5);

// Let’s create a function that accepts an Observable.

const melaenas = pipe (

filter ((n: number) => n % 2 === 0),

map (n => n \* 2)

);

// Let’s create an Observable that will run the filter and map functions

const uneven = mulEvenVals(nums);

// Subscribe to run the combined functions

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

typescript

There is a shorter way to use the pipe function. In the below example, we will explore how we can use the map operator and filter operator using the short form of the pipe() function.

1

2

3

4

5

6

7

8

9

10

import {filter, map} from 'raj’s/operators';

const uneven = of (1, 2, 3, 4, 5)

. pipe (

filter (n => n % 2 === 0),

map (n => n \* 2)

);

// Subscribe to get values

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

typescript

## **Multiple Map Operators**

Now that we understand a little bit about the map operator and pipe functions, let’s now try some complicated examples using multiple map operators.

1

2

3

4

5

6

7

Const myData = of(‘Hello’).pipe(

map(x => {

return ‘${x} world’;

} ),

map(x => ‘${x} of’),

map(x => ‘${x} Angular’)

);

typescript

The output will be ‘Hello world of Angular’. Here, the pipe function will use the reduce function internally. In the reduced code, we can see the string being built up as it passes through each one of the map operators. Eventually, it will produce the string, ‘Hello world of Angular’. The input variable is still the observable returned from of(‘Hello’). With an understanding of how data flows through a single operator, it’s not hard to extend that understanding to multiple operators.

## **Summary**

In this guide, we have learned that operators, like the map and filter, are functions that take in and return observables.

* Each operator exposes a public function, like a map or filter, which we will import from the 'rxjs/operators' library and pass it into the pipe.
* Each operator has an operator class which implements the operator interface so that it can subscribe to other observables.
* Each operator has a subscriber class which contains the logic for that operator (invocation of the projection function for the map, the invocation of the predicate function for the filter, etc.).

We’ve also seen how the pipe is used to compose operators together. Internally, it’s taking the values emitted by the source observable, and reducing it over the list of operators.

## **Conclusion**

In this guide, we have explored how we can use map operators to transform data in Angular. We have also seen how we can use the map operator along with other operators and pipe functions in our application.

You can learn more about subscribing to and unsubscribing from observables in Angular in my guide [Subscribing to and Unsubscribing from Observables](https://app.pluralsight.com/guides/subscribing-to-unsubscribing-from-observables).