Section 13: Understanding Observables: Table of Contents

[170. Module Introduction 1](#_Toc63963531)

[172. Getting Closer to the Core of Observables 2](#_Toc63963532)

[173. Building a Custom Observable 2](#_Toc63963533)

[174. Errors & Completion 3](#_Toc63963534)

[176. Understanding Operators 3](#_Toc63963535)

[177. Subjects 4](#_Toc63963536)

[179. Useful Resources & Links 5](#_Toc63963537)

# 170. Module Introduction

An observable can be thought of as a data source. In our angular project, an observable is basically an object we import from a 3rd-party project, and follows the observable pattern:

In the pattern, you have an observable and an observer. Between then, you have a timeline, and on this timeline multiple events (data packages) can be emitted by the observable.

An observer is your code, i.e.: the subscribe function. As an observer, there are three ways to handle a data packaged: handling the data itself, handling an error, or handling a completion. Each are handled with a hook, but an observable doesn't necessarily have to complete (i.e.: buttons-triggered events)

# 172. Getting Closer to the Core of Observables

The interval method is an observable function that can be imported from the rxjs package. With it, you are able to subscribe to an event that occurs in specified increments. By setting a variable of type Subscription (also imported from rxjs) to the subscribed method, you gain the ability to perform different methods on the variable, such as unsubscribing.

1. private firstObsSubscription: Subscription
3. ngOnInit() {
4. this.firstObsSubscription = interval(1000).subscribe( count => {
5. console.log(count);
6. })
7. }
9. ngOnDestroy() {
10. this.firstObsSubscription.unsubscribe();
11. }

With components that are not included in Angular, the unsubscribe() method needs to be included in the onDestroy lifecycle hook to ensure that multiple observables are not created, producing a memory leak that drains resources.

# 173. Building a Custom Observable

To create a custom observable, the Observable class first needs to be imported from rxjs. An observable is created by setting a new const equal to the Observable.create() method. This method takes a function as an argument, which itself takes an argument that rxjs automatically passes in for us - the observer. This is the listening part that we will need to inform of what's happening.

To do this, we use methods on the observer variable such as next(), error(), or complete(). next() is used to emit a new value, while error() is used to throw and error, and complete() would let the observer know that we are done. The below example works the same as the interval observable used prior.

1. const customIntervalObservable = Observable.create(observer => {
2. let count = 0;
3. setInterval(() => {
4. observer.next(count);
5. count++;
6. }, 1000);
7. });

# 174. Errors & Completion

In a custom observable, if the error() method is called, the error specified as an input argument will be thrown. This error can be handled in the subscribe() method by inputting a function that takes the error as the 2nd argument to subscribe. An observable can be completed simply by including some condition that results in the complete() method being called. A function passed as the 3rd argument in the subscribe() method will do something upon completion, but will take no arguments itself. An error will result in the observable cancelling without completion.

1. if (count == 5) {
2. observer.complete();
3. }
4. if (count > 3) {
5. observer.error(new Error('Count is greater 3!'));
6. }
7. this.firstObsSubscription = customIntervalObservable.subscribe(data => {
8. console.log(data);
9. }, error => {
10. console.log(error);
11. alert(error.message);
12. }, () => {
13. console.log('Completed!');
14. })

# 176. Understanding Operators

Operators allow you to transform data that is received from an observable before it reaches the subscription. To add operators, you have to append the pipe() method onto your observable prior to the subscribe() method. There are may operators that can be imported from rxjs/operators, a couple are the map() and filter() functions.

To use operators, you need to enter them as arguments into the pipe() method, in the order that they should appear. Typically filter before map. With filter, you can return a condition that will permit data to reach map and subscribe should the condition be true. With map, you can return transformed data after operating on the transmitted data in some fashion. Ex:

1. this.firstObsSubscription = customIntervalObservable.pipe(
2. filter(data => {
3. return data > 0;
4. }),
5. map((data: number) => {
6. return 'Round: ' + (data + 1);
7. }))
8. .subscribe(...)

# 177. Subjects

A Subject (imported from rxjs) is used in place of an EventEmitter definition. and when calling the subject, the next() method is used in place of the emit() method.

A subject is similar to an observable in that it is an object you can subscribe to. However, it is more active since you can call next from outside, whereas an observable is more passive (you always wrap a callback or event). Since this allows you to trigger the Subject from the code, it is more useful for instances with no passive event source, i.e button trigger.

A couple extra notes are that subjects need to be unsubscribed from since they are observables, and they can only replace EventEmitters in cross-component communication, not with @Output.

service:

1. export class UserService {
2. activatedEmitter = new Subject<boolean>();
3. }

component:

1. onActivate() {
2. this.userService.activatedEmitter.next(true);
3. }

# 179. Useful Resources & Links

Useful Resources:

* Official Docs: <https://rxjs-dev.firebaseapp.com/>
* RxJS Series: <https://academind.com/learn/javascript/understanding-rxjs/>
* Updating to RxJS 6: <https://academind.com/learn/javascript/rxjs-6-what-changed/>