RxJS: Understanding Subjects

Nicholas Jamieson Follow
Feb 2, 2018 · 5 min read

Photo by Matt Artz on Unsplash

I see a lot of questions about subjects on Stack Overflow. Recently, I saw one that asked how an `AsyncSubject` should be used. The question prompted me to write this article to show why the various types of subjects are necessary and how they are used in RxJS itself.

What's the use case for subjects?

In his article *On the Subject of Subjects*, Ben Lesh states that:

> *... [multicasting] is the primary use case for Subjects in RxJS.*

We'll look at multicasting in more detail later in the article, but for now it's enough to know that it involves taking the notifications from a single, source observable and forwarding them to one or more destination observers.

This connecting of observers to an observable is what subjects are all about. They're able to do it because subjects themselves are both observers and observables.

How can subjects be used?

Let's use an Angular component as an example: an `awesome-component`. Our component does some awesome stuff and has an internal observable that emits values as the user interacts with the component.

To enable parent components to connect to the observable, the `awesome-component` accepts an `observer` input property—which it subscribes to the observable. That means the parent could connect to the observable by specifying an observer, like this:

```
@Component({
  selector: "parent-component",
  template: `<awesome-component [observer]="observer">
})
export class ParentComponent {
  public observer: PartialObserver<any>;
  constructor() {
    this.observer = {
      next(value) { /* do something with the value */
```

With the observer wired up, the parent is connected and receives values from the `awesome-component`. However, this is essentially the same as if the `awesome-component` had emitted its values using an output event. So why not use an event?

Observables have the advantage of being easy to manipulate. For example, it's easy to add filtering and debouncing just by applying a few operators. But the parent component has an observer—not an observable—so how can we apply operators?

Subjects are both observers and observables, so if we create a `Subject`, it can be passed to the `awesome-component` (as an observer) and can have debouncing applied to it (as an observable), like this:

```
@Component({
  selector: "parent-component",
  template: `<awesome-component [observer]="observer">
})
export class ParentComponent {
  public observer: PartialObserver<any>;
  private _subject: Subject<any>;
  constructor() {
    this._subject = new Subject<any>();
    this._subject.pipe(
      debounceTime(1000),
```

The subject connects the do-something-with-the-value observer with the `awesome-component` observable, but with the parent component's choice of operators applied.

Composing different observables

By using a `Subject` to compose an observable, the `awesome-component` can be used in different ways by different components. For example, another component might be interested in only the last-emitted value. That component could use the `last` operator:

```
@Component({
  selector: "another-component",
  template: `<awesome-component [observer]="observer">
})
export class AnotherComponent {
  public observer: PartialObserver<any>;
  private _subject: Subject<any>;
  constructor() {
    this._subject = new Subject<any>();
    this._subject.pipe(
      last()
```

Interestingly, there is another way that component could choose to receive only the last-emitted value from the `awesome-component` : it could use a different type of subject. An `AsyncSubject` emits only the last-received value, so an alternative implementation would be:

```
@Component({
  selector: "another-component",
  template: `<awesome-component [observer]="observer">
})
export class AnotherComponent {
  public observer: PartialObserver<any>;
  private _subject: Subject<any>;
  constructor() {
    this._subject = new AsyncSubject<any>();
```

If using an `AsyncSubject` is equivalent to composing the observable using a `Subject` and the `last` operator, why complicate RxJS with the `AsyncSubject` class?

Well, it's because subjects are primarily for multicasting.

The two are equivalent here, because there is a single subscriber—the do-something-with-the-value observer. In a multicasting situation, there can be multiple subscribers and applying the `last` operator to a `Subject` won't effect the same behaviour as an `AsyncSubject` for late subscribers.

Let's have a closer look at multicasting.

How are subjects used in RxJS?

The core of RxJS's multicasting infrastructure is implemented using a single operator: `multicast`. The `multicast` operator is applied to a source observable, takes a subject (or a factory that creates a subject) and returns an observable composed from the subject.

The `multicast` operator is somewhat like the `awesome-component` in our examples: we can obtain an observable that exhibits different behaviour simply by passing a different type of subject.

When a basic `Subject` is passed to `multicast`:

- subscribers to the multicast observable receive the source's `next`, `error` and `complete` notifications; and
- late subscribers—i.e. those that subscribe after an `error` or `complete` notification has occurred—receive the `error` or `complete` notification.

It's important to note that unless `multicast` is passed a factory, late subscribers don't effect another subscription to the source.

To compose a multicast observable that forwards the source observable's last-emitted `next` notification to *all* subscribers, it's not enough to apply the `last` operator to a multicast observable that was created using a `Subject`. Late subscribers to such an observable won't receive the last-emitted `next` notification; they will receive only the `complete` notification.

For late subscribers to receive the last-emitted `next` notification, the notification needs to be stored in the subject's state. That's what the `AsyncSubject` does and that's why the `AsyncSubject` class is necessary.

What about the other subject classes?

There are two other subject variants: `BehaviorSubject` and `ReplaySubject`.

To understand the `BehaviorSubject`, let's have a look at another component-based example:

```
@Component({
  selector: "parent-component",
  template: `<awesome-component [observer]="observer">
})
export class ParentComponent {
  public observer: PartialObserver<any>;
  private _subject: Subject<any>;
  constructor() {
    this._subject = new Subject<any>();
    this._subject.pipe(
      startWith("awesome")
```

Here, the parent component connects to the `awesome-component` using a `Subject` and applies the `startWith` operator. Using `startWith` ensures that the parent receives the value `"awesome"` upon subscription, followed by the values emitted by the `awesome-component` —whenever they happen to be emitted.

In the same way that an `AsyncSubject` replaced the use of a `Subject` and the `last` operator, a `BehaviorSubject` could replace the use of a `Subject` and the `startWith` operator—with the `BehaviorSubject`'s

constructor taking the value that would otherwise have been passed to `startWith`.

However, using a `Subject` and the `startWith` operator won't effect the desired behaviour in a multi-subscriber situation. The first subscriber will see the expected behaviour, but subsequent subscribers will always receive the `startWith` value—even if the source has already emitted a value.

If a `BehaviorSubject` is used, subsequent subscribers will receive the initial value if the source has not yet emitted or the most-recently-emitted value if it has. This is possible because the `BehaviorSubject` stores the value in its state.

There is no single-subscriber analogy for the `ReplaySubject`, as the concept of replaying already received notifications is inherently multi-subscriber. To facilitate the replaying of notifications to subsequent subscribers, the `ReplaySubject` stores the notifications in its state.

So how do you use these subjects?

Now that we've seen what the various subjects do and why they are necessary, how should they be used? Well, it's quite likely that the only subject class you will ever need to use will be a `Subject`.

A `Subject` works just fine for connecting an observer to an observable. And for the multicasting situations, there is an alternative.

RxJS contains multicasting operators that use the various subject classes and in the same way that I favour using RxJS observable creators (like `fromEvent`) over calls to `Observable.create`, for multicasting situations I favour using RxJS operators over explicit subjects:

- `publish` or `share` can be used instead of a `Subject`;
- `publishBehaviour` can be used instead of a `BehaviorSubject`;
- `publishLast` can be used instead of an `AsyncSubject`; and
- `publishReplay` or `shareReplay` can be used instead of a `ReplaySubject`.

The `publish` and `share` operators are covered in more detail in my articles:

- *RxJS: Understanding the publish and share Operators*; and
- *RxJS: How to Use refCount*.