Using observables to pass values

Observables provide support for passing messages between parts of your application. They are used frequently in Angular and are a technique for event handling, asynchronous programming, and handling multiple values.

The observer pattern is a software design pattern in which an object, called the *subject*, maintains a list of its dependents, called *observers*, and notifies them automatically of state changes. This pattern is similar (but not identical) to the publish/subscribe design pattern.

Observables are declarative—that is, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe.

An observable can deliver multiple values of any type—literals, messages, or events, depending on the context. The API for receiving values is the same whether the values are delivered synchronously or asynchronously. Because setup and teardown logic are both handled by the observable, your application code only needs to worry about subscribing to consume values, and when done, unsubscribing. Whether the stream was keystrokes, an HTTP response, or an interval timer, the interface for listening to values and stopping listening is the same.

Because of these advantages, observables are used extensively within Angular, and for application development as well.

Basic usage and terms

As a publisher, you create an `Observable` instance that defines a *subscriber* function. This is the function that is executed when a consumer calls the `subscribe()` method. The subscriber function defines how to obtain or generate values or messages to be published.

To execute the observable you have created and begin receiving notifications, you call its `subscribe()` method, passing an *observer*. This is a JavaScript object that defines the handlers for the notifications you receive. The `subscribe()` call returns a `Subscription` object that has an `unsubscribe()` method, which you call to stop receiving notifications.

Here's an example that demonstrates the basic usage model by showing how an observable could be used to provide geolocation updates.

Defining observers

A handler for receiving observable notifications implements the `Observer` interface. It is an object that defines callback methods to handle the three types of notifications that an observable can send:

Notification type	Description
`next`	Required. A handler for each delivered value. Called zero or more times after execution starts.
`error`	Optional. A handler for an error notification. An error halts execution of the observable instance.
`complete`	Optional. A handler for the execution-complete notification. Delayed values can continue to be delivered to the next handler after execution is complete.

An observer object can define any combination of these handlers. If you don't supply a handler for a notification type, the observer ignores notifications of that type.

Subscribing

An `Observable` instance begins publishing values only when someone subscribes to it. You subscribe by calling the `subscribe()` method of the instance, passing an observer object to receive the notifications.

In order to show how subscribing works, we need to create a new observable. There is a constructor that you use to create new instances, but for illustration, we can use some methods from the RxJS library that create simple observables of frequently used types:

- `of(...items)`—Returns an `Observable` instance that synchronously delivers the values provided as arguments.
- `from(iterable)`—Converts its argument to an `Observable` instance. This method is commonly used to convert an array to an observable.

Here's an example of creating and subscribing to a simple observable, with an observer that logs the received message to the console:

Alternatively, the `subscribe()` method can accept callback function definitions in line, for `next`, `error`, and `complete` handlers. For example, the following `subscribe()` call is the same as the one that specifies the predefined observer:

In either case, a `next` handler is required. The `error` and `complete` handlers are optional.

Note that a `next()` function could receive, for instance, message strings, or event objects, numeric values, or structures, depending on context. As a general term, we refer to data published by an observable as a *stream*. Any type of value can be represented with an observable, and the values are published as a stream.

Creating observables

Use the `Observable` constructor to create an observable stream of any type. The constructor takes as its argument the subscriber function to run when the observable's `subscribe()` method executes. A subscriber function receives an `Observer` object, and can publish values to the observer's `next()` method.

For example, to create an observable equivalent to the `of(1, 2, 3)` above, you could do something like this:

To take this example a little further, we can create an observable that publishes events. In this example, the subscriber function is defined inline.

Now you can use this function to create an observable that publishes keydown events:

Multicasting

A typical observable creates a new, independent execution for each subscribed observer. When an observer subscribes, the observable wires up an event handler and delivers values to that observer. When a second observer subscribes, the observable then wires up a new event handler and delivers values to that second observer in a separate execution.

Sometimes, instead of starting an independent execution for each subscriber, you want each subscription to get the same values—even if values have already started emitting. This might be the case with something like an observable of clicks on the document object.

Multicasting is the practice of broadcasting to a list of multiple subscribers in a single execution. With a multicasting observable, you don't register multiple listeners on the document, but instead re-use the first listener and send values out to each subscriber.

When creating an observable you should determine how you want that observable to be used and whether or not you want to multicast its values.

Let's look at an example that counts from 1 to 3, with a one-second delay after each number emitted.

Notice that if you subscribe twice, there will be two separate streams, each emitting values every second. It looks something like this:

Changing the observable to be multicasting could look something like this:

Multicasting observables take a bit more setup, but they can be useful for certain applications. Later we will look at tools that simplify the process of multicasting, allowing you to take any observable and make it multicasting.

Error handling

Because observables produce values asynchronously, try/catch will not effectively catch errors. Instead, you handle errors by specifying an `error` callback on the observer. Producing an error also causes the observable to clean up subscriptions and stop producing values. An observable can either produce values (calling the `next` callback), or it can complete, calling either the `complete` or `error` callback.

myObservable.subscribe({ next(num) { console.log('Next num:' + num)}, error(err) { console.log('Received an error:' + err)} });

Error handling (and specifically recovering from an error) is covered in more detail in a later section.