Using observables to pass values

Contents ›

Observables provide support for passing messages between parts of your application. They are used frequently in Angular and are the recommended 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 are recommended for app 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.

Observe geolocation updates

```
// Create an Observable that will start listening to geolocation updates
// when a consumer subscribes.
const locations = new Observable((observer) => {
  let watchId: number;

  // Simple geolocation API check provides values to publish
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition((position: Position) => {
      observer.next(position);
    }, (error: PositionError) => {
      observer.error(error);
    });
  } else {
    observer.error('Geolocation not available');
  }

  // When the consumer unsubscribes, clean up data ready for next subscription.
  return {
    unsubscribe() {
      navigator.geolocation.clearWatch(watchId);
    }
  };
});

// Call subscribe() to start listening for updates.
const locationsSubscription = locations.subscribe({
  next(position) {
    console.log('Current Position: ', position);
  },
  error(msg) {
    console.log('Error Getting Location: ', msg);
  }
});

// Stop listening for location after 10 seconds
setTimeout(() => {
  locationsSubscription.unsubscribe();
}, 10000);
```

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:

Subscribe using observer

```
// Create simple observable that emits three values
const myObservable = of(1, 2, 3);

// Create observer object
const myObserver = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};
```

```
// Execute with the observer object
myObservable.subscribe(myObserver);
// Logs:
// Observer got a next value: 1
// Observer got a next value: 2
// Observer got a next value: 3
// Observer got a complete notification
```

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:

Subscribe with positional arguments

```
myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

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:

Create observable with constructor

```
// This function runs when subscribe() is called
function sequenceSubscriber(observer) {
  // synchronously deliver 1, 2, and 3, then complete
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  // unsubscribe function doesn't need to do anything in this
  // because values are delivered synchronously
  return {unsubscribe() {}};
}
```

```
// Create a new Observable that will deliver the above sequence
const sequence = new Observable(sequenceSubscriber);

// execute the Observable and print the result of each notification
sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});

// Logs:
// 1
// 2
// 3
// Finished sequence
```

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

Create with custom fromEvent function

```
function fromEvent(target, eventName) {
  return new Observable((observer) => {
    const handler = (e) => observer.next(e);

    // Add the event handler to the target
    target.addEventListener(eventName, handler);

    return () => {
      // Detach the event handler from the target
      target.removeEventListener(eventName, handler);
    };
  });
}
```

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

Use custom fromEvent function

```
const ESC_KEY = 27;
const nameInput = document.getElementById('name') as HTMLInputElement;

const subscription = fromEvent(nameInput, 'keydown')
  .subscribe((e: KeyboardEvent) => {
    if (e.keyCode === ESC_KEY) {
      nameInput.value = '';
```

```
    }
  });
```

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.

Create a delayed sequence

```
function sequenceSubscriber(observer) {
  const seq = [1, 2, 3];
  let timeoutId;

  // Will run through an array of numbers, emitting one value
  // per second until it gets to the end of the array.
  function doInSequence(arr, idx) {
    timeoutId = setTimeout(() => {
      observer.next(arr[idx]);
      if (idx === arr.length - 1) {
        observer.complete();
      } else {
        doInSequence(arr, ++idx);
      }
    }, 1000);
  }

  doInSequence(seq, 0);

  // Unsubscribe should clear the timeout to stop execution
  return {unsubscribe() {
    clearTimeout(timeoutId);
  }};
```

```
}

// Create a new Observable that will deliver the above sequence
const sequence = new Observable(sequenceSubscriber);

sequence.subscribe({
  next(num) { console.log(num); },
  complete() { console.log('Finished sequence'); }
});

// Logs:
// (at 1 second): 1
// (at 2 seconds): 2
// (at 3 seconds): 3
// (at 3 seconds): Finished sequence
```

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

Two subscriptions

```
// Subscribe starts the clock, and will emit after 1 second
sequence.subscribe({
  next(num) { console.log('1st subscribe: ' + num); },
  complete() { console.log('1st sequence finished.'); }
});

// After 1/2 second, subscribe again.
setTimeout(() => {
  sequence.subscribe({
    next(num) { console.log('2nd subscribe: ' + num); },
    complete() { console.log('2nd sequence finished.'); }
  });
}, 500);

// Logs:
// (at 1 second): 1st subscribe: 1
// (at 1.5 seconds): 2nd subscribe: 1
// (at 2 seconds): 1st subscribe: 2
// (at 2.5 seconds): 2nd subscribe: 2
// (at 3 seconds): 1st subscribe: 3
// (at 3 seconds): 1st sequence finished
// (at 3.5 seconds): 2nd subscribe: 3
// (at 3.5 seconds): 2nd sequence finished
```

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

Create a multicast subscriber

```
function multicastSequenceSubscriber() {
  const seq = [1, 2, 3];
  // Keep track of each observer (one for every active subscription)
  const observers = [];
  // Still a single timeoutId because there will only ever be one
  // set of values being generated, multicasted to each subscriber
  let timeoutId;

  // Return the subscriber function (runs when subscribe()
  // function is invoked)
  return (observer) => {
    observers.push(observer);
    // When this is the first subscription, start the sequence
    if (observers.length === 1) {
      timeoutId = doSequence({
        next(val) {
          // Iterate through observers and notify all subscriptions
          observers.forEach(obs => obs.next(val));
        },
        complete() {
          // Notify all complete callbacks
          observers.slice(0).forEach(obs => obs.complete());
        }
      }, seq, 0);
    }

    return {
      unsubscribe() {
        // Remove from the observers array so it's no longer notified
        observers.splice(observers.indexOf(observer), 1);
        // If there's no more listeners, do cleanup
        if (observers.length === 0) {
          clearTimeout(timeoutId);
        }
      }
    };
  };
}

// Run through an array of numbers, emitting one value
// per second until it gets to the end of the array.
function doSequence(observer, arr, idx) {
  return setTimeout(() => {
    observer.next(arr[idx]);
```

```
    if (idx === arr.length - 1) {
      observer.complete();
    } else {
      doSequence(observer, arr, ++idx);
    }
  }, 1000);
}

// Create a new Observable that will deliver the above sequence
const multicastSequence = new Observable(multicastSequenceSubscriber());

// Subscribe starts the clock, and begins to emit after 1 second
multicastSequence.subscribe({
  next(num) { console.log('1st subscribe: ' + num); },
  complete() { console.log('1st sequence finished.'); }
});

// After 1 1/2 seconds, subscribe again (should "miss" the first value).
setTimeout(() => {
  multicastSequence.subscribe({
    next(num) { console.log('2nd subscribe: ' + num); },
    complete() { console.log('2nd sequence finished.'); }
  });
}, 1500);

// Logs:
// (at 1 second): 1st subscribe: 1
// (at 2 seconds): 1st subscribe: 2
// (at 2 seconds): 2nd subscribe: 2
// (at 3 seconds): 1st subscribe: 3
// (at 3 seconds): 1st sequence finished
// (at 3 seconds): 2nd subscribe: 3
// (at 3 seconds): 2nd sequence finished
```

> 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.

Version 10.0.10-local+sha.b32126c335.