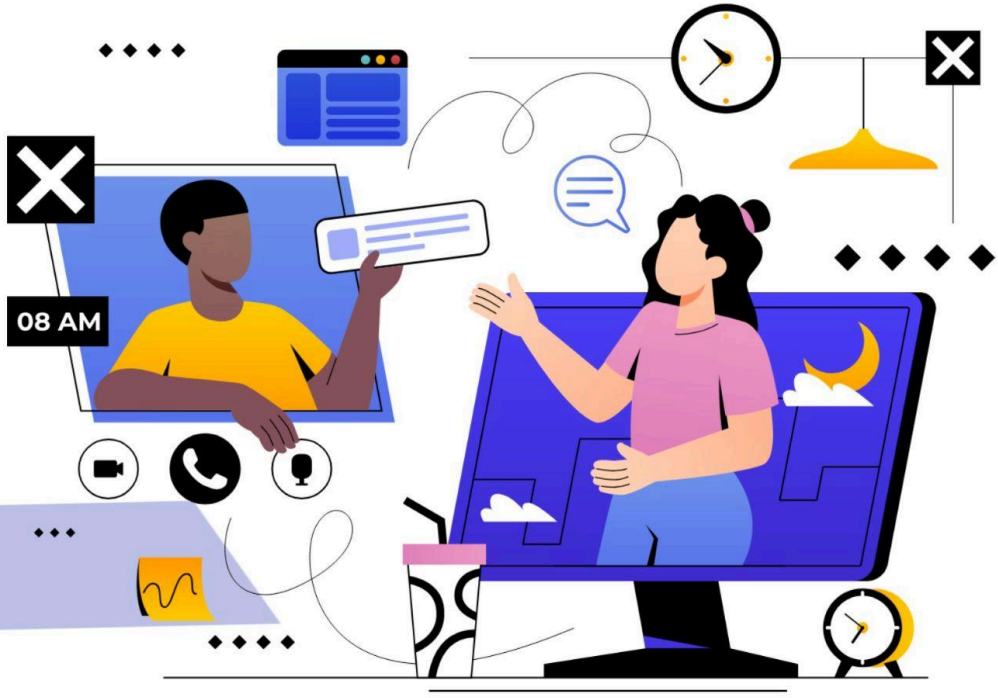
Angular Observables and Promises

A Practical Guide to Asynchronous Programming



Abdelfattah Ragab

Angular Observables and Promises

A Practical Guide to Asynchronous Programming

Abdelfattah Ragab

Introduction

Welcome to the book “Angular Observables and Promises: A Practical Guide to Asynchronous Programming”.
In this book, I explain how to use observables and promises effectively for asynchronous programming.
I show you practical scenarios and explain when you should use them and which operators you need to use.
I will give you best practices with important pointers that, when used correctly, can make all the difference and have a big impact on performance.
There are also things you should be aware of and avoid when working with observables, otherwise performance can be affected.
By the end of this book, you will be able to use Observables and Promises in your Angular application and handle all kinds of scenarios.
Let us get started.

What are Signals?

Signals are a new feature in Angular that improves the reactivity of the framework and the detection of changes.
Signals are used for **synchronous** operations, so you should use them to manage state and update the user interface efficiently.

I mentioned them at the beginning to illustrate the role they play in Angular. I will not go into them in this book, as synchronous operations are not the topic of this book. I will mainly focus on the asynchronous operations.

What is RxJS?

RxJS (Reactive Extensions for JavaScript) is a powerful reactive programming library that uses observables to manage asynchronous data streams. It enables developers to work with asynchronous operations in a declarative style, making it easier to handle complex data streams and events.

What are Observables?

Observables are part of the RxJS library that allow you to handle asynchronous **data streams**.

What are Promises?

A promise in JavaScript is an object that represents the eventual completion or failure of an asynchronous operation.
A promise handles **one single value**.

`new Promise`

Most of the time, Promises and Observables are created for you by libraries, http, Apis and so on. But if you want to create one yourself, you can use the constructor. Let's look at how to create a promise with the constructor. It takes a function with two parameters: **resolve** and **reject**. Call resolve if the asynchronous operation succeeds, and reject if it fails.

```
myPromise = new Promise((resolve,
reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve('Resolved');
    } else {
      reject('Rejected');
    }
  }, 3000);
});
```

`Promise.resolve()`

You can create a successful promise using `Promise.resolve()`.

```
const promise=
Promise.resolve('Resolved');
```

`Promise.reject()`

Similarly, you can create a rejected promise using `Promise.reject()`.

```
const promise =
Promise.reject('Rejected');
```

Using Promises

Use `.then()` to handle the resolved state and `.catch()` to handle the rejected state.

```
this.promise
  .then((result: any) => {
    console.log(result);
  })
  .catch((error: any) => {
    console.error(error);
  });
```

Async/Await

Async/Await offers a more readable way of working with asynchronous code than is possible with `.then()` and `.catch()`.
It consists of two parts in order to function:

- Declare the outer function as async. Simply add the word `async` in front of the function name.
- Use the word `await` when you call the promise.

```
async fetchData() {
  const result = await
this.productService.getProduct(1);
  console.log(result);
  return result;
}
```

new Observable

You can create an observable using the constructor:

```
myObservable = new
Observable((subscriber) => {
  subscriber.next('First value');
  subscriber.next('Second value');

  subscriber.complete();
});
```

Unlike Promises, which are created within JavaScript and do not require imports, you need to import Observables from `'rxjs'`.

`of`

To create an observable that outputs the specified values.

```
myObservable = of(1, 2, 3);
```

`from`

To convert various data types into observables.

```
myObservable = from(myPromise)
```

`HTTP` returns observables

All `HttpClient` methods return observables.

```
myObservable =
this.http.get('https://api.examp..');
```

Subscribe to an Observable

To subscribe to an observable, use the `subscribe` method, which takes up to three arguments:

- **`next`**: called whenever the observable outputs a new value.
- **`error`**: called in case of errors.

- **`complete`**: called when the observable is completed.

```
this.myObservable.subscribe({
  next: (value: any) => {
    console.log(value);
  },
  complete: () => {
    console.log('Observable
completed.');
  },
  error: (e: any) => {
    console.log(e);
  },
});
```

unsubscribe

When you subscribe to an observable, it returns a subscription object. This object can be used to unsubscribe from the observable, which is important to avoid memory leaks, especially in Angular components.

```
this.subscription =
this.myObservable.subscribe(..);
```

Unsubscribe when the component is destroyed.

```
ngOnDestroy(): void {
  if (this.subscription)
this.subscription.unsubscribe();
}
```

Cold Observables

Cold observables are a type of observable in RxJS that **only output values when a subscriber subscribes to them**. This means that data production is directly linked to the subscription process. Each subscriber to a cold observable receives their own independent execution of the observable, which can lead to different results depending on when they subscribe to it.
Common examples of cold observables are HTTP requests.

Hot Observables

Hot observables are a type of observable in RxJS that **generate values regardless of whether there are subscribers**. This means that data is generated independently of the subscription process.

All subscribers share the same version of the observable. This means that if several subscribers establish a connection to a hot observable, they all receive the same output values from this point onwards, but no values that were output before they logged in.

Common examples of hot observables are events such as mouse movements or keystrokes.

```
hotObservable = new Subject();
```