Being Reactive Using Observables and RxJS

Handling asynchronous information is a common task in our everyday lives as developers. **Reactive programming** is a programming paradigm that helps us consume, digest, and transform asynchronous information using data streams. **RxJS** is a JavaScript library that provides methods to manipulate data streams using **observables**.

Angular provides an unparalleled toolset to help us when it comes to working with asynchronous data. Observable streams are at the forefront of this toolset, giving developers a rich set of capabilities when creating Angular applications. The core of the Angular framework is lightly dependent on RxJS. Other Angular packages, such as the router and the HTTP client, are more tightly coupled with observables.

In this chapter, we will learn about the following concepts:

- Strategies for handling asynchronous information
- Reactive programming in Angular
- The RxJS library
- Subscribing to observables
- Unsubscribing from observables

Technical requirements

The chapter contains various code samples to walk you through the concept of observables and RxJS. You can find the related source code in the `ch07` folder of the following GitHub repository:

https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition

Strategies for handling asynchronous information

We manage data asynchronously in different forms, such as consuming data from a backend API or reading contents from the local filesystem. Consuming information from an API is a typical operation in our daily development workflow. We consume data over HTTP all the time, such as when authenticating users by sending out credentials to an authentication service. We also use HTTP when fetching the latest tweets in our favorite Twitter widget. Modern mobile devices have introduced a unique way of consuming remote services. They defer requests and response consumption until mobile connectivity is available. Responsivity and availability have

become a big deal.

Although internet connections are high-speed nowadays, response time is always involved when serving such information. Thus, as we will see in the following sections, we put in place mechanisms to handle states in our applications transparently for the end user.

Shifting from callback hell to promises

Sometimes, we might need to build functionalities in our application that change its state asynchronously once some time has elapsed. We must introduce code patterns such as the **callback pattern** to handle this deferred change in the application state.

In a callback, the function that triggers asynchronous action accepts another function as a parameter. The function is called when the asynchronous operation has been completed. Let's see how to use a callback through an example:

1. First, run the following Angular CLI command to create a new Angular application:

```
ng new my-app --defaults
```

   In the preceding command, the `--defaults` option instructs the Angular CLI to create the Angular project

with default values for routing and styling.

2. Open the `app.component.ts` file and create a `setTitle` property to change the `title` property of the component. Notice that it returns an arrow function because we are going to use it as a callback to another method:

```
private setTitle = () => {
  this.title = 'Learning Angular';
}
```

3. Next, create a `changeTitle` method that calls another method, named, by convention, `callback`, after 2 seconds:

```
private changeTitle(callback: Function) {
  setTimeout(() => {
    callback();
  }, 2000);
}
```

4. Finally, add a `constructor` in the component and call the `changeTitle` method, passing the `setTitle` property as a parameter:

```
constructor() {
  this.changeTitle(this.setTitle);
}
```

In the preceding snippet, we use the `setTitle` property without parentheses because we pass function signatures and not actual function calls when we use callbacks.

If we run the Angular application using the `ng serve` command, we see that the `title` property changes after 2 seconds. The problem with the pattern we just described is that the code can become confusing and cumbersome as we introduce more nested callbacks.

Consider the following scenario where we need to drill down into a folder hierarchy to access photos on a device:

```
getRootFolder(folder => {
  getAssetsFolder(folder, assets => {
    getPhotos(assets, photos => {});
  });
});
```

We depend on the previous asynchronous call and the data it brings back before we can do the next call. We must execute a method inside a callback that executes another method with a callback. The code quickly ends up looking horrible and complicated, which leads to a situation known as **callback hell**.

We can avoid callback hell using **promises**. Promises introduce a new way of envisioning asynchronous data management by conforming to a neater and more solid interface. Different asynchronous operations can be chained at the same level and even be split and returned from other functions.

To better understand how promises work, let's refactor our previous callback example:

1. Create a new method in the `AppComponent` class named `onComplete` that returns a `Promise` object. A promise can either be resolved or rejected. The `resolve` parameter indicates that the promise was completed successfully and optionally returns a result:

```
private onComplete() {
  return new Promise<void>(resolve => {
  });
}
```

2. Introduce a timeout of 2 seconds in the promise so that it resolves after this time has elapsed:

```
private onComplete() {
  return new Promise<void>(resolve => {
    setTimeout(() => {
```

```
      resolve();
    }, 2000);
  });
}
```

3. Now, replace the `changeTitle` call in the `constructor` with the promise-based method. To execute a method that returns a promise, we invoke the method and chain it with the `then` method:

```
constructor() {
  this.onComplete().then(this.setTitle);
}
```

We do not notice any significant difference if we rerun the Angular application. The real value of promises lies in the simplicity and readability afforded to our code. We could now refactor the previous folder hierarchy example accordingly:

```
getRootFolder()
  .then(getAssetsFolder)
  .then(getPhotos);
```

The chaining of the `then` method in the preceding code shows how we can line up one asynchronous call after another. Each previous asynchronous call passes its result in the upcoming asynchronous method.

Promises are compelling, but why do we need another paradigm? Sometimes we might need to produce a response output that follows a more complex digest process or even cancel the whole process. We cannot accomplish such behavior with promises because they are triggered as soon as they're instantiated. In other words, promises are not lazy. On the other hand, the possibility of tearing down an asynchronous operation after it has been fired but not completed yet can become quite handy in specific scenarios. Promises allow us to resolve or reject an asynchronous operation, but sometimes we might want to abort everything before getting to that point.

On top of that, promises behave as one-time operations. Once they are resolved, we cannot expect to receive any further information or state change notification unless we rerun everything from scratch. Moreover, we sometimes need a more proactive implementation of asynchronous data handling, which is where observables come into the picture. To summarize the limitations of promises:

- They cannot be canceled.
- They are immediately executed.
- They are one-time operations; there is no easy way to retry them.
- They respond with only one value.

Observables in a nutshell

An observable is an object that maintains a list of dependents, called **observers**, and informs them about state

and data changes by emitting events asynchronously. To do so, the observable implements all the machinery it needs to produce and emit such events. It can be fired and canceled at any time, regardless of whether it has emitted the expected data already.

Observers need to subscribe to an observable to be notified and react to reflect the state change. This pattern, known as the **observer pattern**, allows concurrent operations and more advanced logic. These observers, also known as **subscribers**, keep listening to whatever happens in the observable until it is destroyed. We can see all this with more transparency in an actual example:

1. Replace `setTimeout` with `setInterval` in the `onComplete` method that we covered previously:

```
private onComplete() {
  return new Promise<void>(resolve => {
    setInterval(() => {
      resolve();
    }, 2000);
  });
}
```

   The promise will now resolve repeatedly every 2 seconds.

2. Modify the `setTitle` property to append the current timestamp in the `title` property of the component:

```
private setTitle = () => {
  const timestamp = new Date().getMilliseconds();
  this.title = `Learning Angular (${timestamp})`;
}
```

3. Run the Angular application, and you will notice that the timestamp is set only once after 2 seconds and never changes again. The promise resolves itself, and the entire asynchronous event terminates at that very moment. It is not the desired behavior for our application so let's fix it using observables!
4. Import the `Observable` artifact from the `rxjs` npm package:

```
import { Observable } from 'rxjs';
```

5. Create a component property named `title$` that creates an `Observable` object. The constructor of an observable accepts an `observer` object as a parameter. The observer is an arrow function that contains the business logic that will be executed when someone uses the observable. Call the `next` method of the `observer` every 2 seconds to indicate a data or application state change:

```
title$ = new Observable(observer => {
  setInterval(() => {
    observer.next();
  }, 2000);
});
```

When we define an observable variable, we tend to append the `$` sign to the variable name. It is a convention that we follow to identify observables in our code efficiently and quickly.

6. Modify the `constructor` of the component to use the newly created `title$` property:

```
constructor() {
  this.title$.subscribe(this.setTitle);
}
```

We use the `subscribe` method to register to the `title$` observable and get notified of any changes. If we do not call this method, the `setTitle` method will never execute.

An observable will not do anything unless a subscriber subscribes to it.

If you run the application, you will notice that the timestamp now changes every 2 seconds. Congratulations! You have entered the world of observables and reactive programming!

Observables return a stream of events, and our subscribers receive prompt notifications of those events so that they can act accordingly. They do not perform an asynchronous operation and terminate (although we can configure them to do so) but start a stream of ongoing events to which we can subscribe.

That's not all, however. This stream can combine many operations before hitting observers subscribed to it. Just

as we can manipulate arrays with methods such as `map` or `filter` to transform them, we can do the same with the stream of events emitted by observables. It is a pattern known as **reactive programming**, and Angular makes the most of this paradigm to handle asynchronous information.

Reactive programming in Angular

The observer pattern stands at the core of what we know as reactive programming. The most basic implementation of a reactive script encompasses several concepts that we need to become familiar with:

- An observable
- An observer
- A timeline
- A stream of events
- A set of composable operators

Sound daunting? It isn't. The big challenge here is to change our mindset and learn to think reactively, which is the primary goal of this section.

Reactive programming entails applying asynchronous subscriptions and transformations to observable streams of events.

Let's explain it through a more descriptive example. Think about an interaction device such as a keyboard. It has keys that the user presses. Each one of those keystrokes triggers a specific keyboard event, such as `keyUp`. The keyUp event features a wide range of metadata, including—but not limited to—the numeric code of the specific key the user pressed at a given moment. As the user continues hitting keys, more keyUp events are triggered and piped through an imaginary timeline that should look like the following diagram:

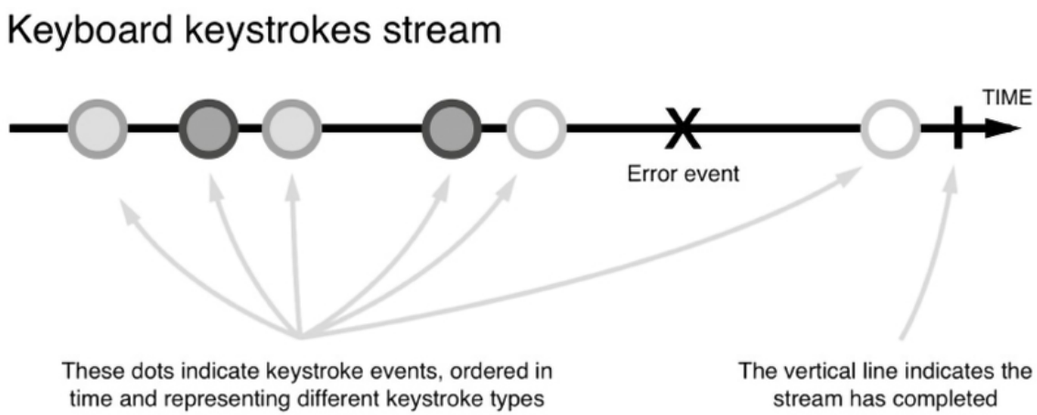

Figure 7.1: Timeline of keystroke events

The timeline is a continuous stream of data where the keyUp event can happen at any time; after all, the user decides when to press those keys. Recall the example with observables from the previous section. That code could notify an observer that every 2 seconds, another value was emitted. What's the difference between that code and our keyUp events? Nothing. We know how often a timer interval is triggered. In the case of keyUp events, we don't know because it is not under our control. But that is the only difference, which means keyUp events can also be considered observables. Let's try to explain it further by implementing a key logger in our app:

1. Create a new Angular component with the name `key-logger`:

```
ng generate component key-logger
```

2. Open the `key-logger.component.html` file and replace its content with the following HTML template:

```
<input type="text" #keyContainer>
You pressed: {{keys}}
```

   In the preceding template, we added an `<input>` HTML element and attached the `keyContainer` template reference variable.

   A template reference variable can be added to any HTML element, not just components.

We also display a `keys` component property representing all the keyboard keys the user has pressed.

3. Open the `key-logger.component.ts` file and import the `OnInit`, `ViewChild`, and `ElementRef` artifacts from the `@angular/core` npm package:

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
```

4. Create the following properties in the `KeyLoggerComponent` class:

```
@ViewChild('keyContainer', { static: true }) input: ElementRef | undefined;
keys = '';
```

   The `input` property is used to query the `<input>` HTML element using the `keyContainer` template reference variable. The second parameter of the `@ViewChild` decorator is an object with a `static` property. The `static` property indicates whether the element we want to query will be available during component initialization. In our case, the `<input>` element is already present on the DOM, so we set its value to `true`. However, there are cases where an HTML element is not present initially, such as when using the `NgIf` directive to add it conditionally. In that case, instead of setting its value to `false`, we can remove the second parameter of the `@ViewChild` decorator completely.

5. Add the following `import` statement to import the `fromEvent` artifact from the `rxjs` npm package:

```
import { fromEvent } from 'rxjs';
```

The RxJS library has a variety of helpful artifacts, called **operators**, that we can use with observables. One of them is the `fromEvent` operator, which creates an observable from the DOM event of a native HTML element.

6. Implement the `ngOnInit` method from the `OnInit` interface to listen for `keyup` events in the `<input>` element and keep pressed keys in the `keys` property:

```
export class KeyLoggerComponent implements OnInit {
  @ViewChild('keyContainer', { static: true }) input: ElementRef | undefined;
  keys = '';
  ngOnInit(): void {
    const logger$ = fromEvent<KeyboardEvent>(this.input?.nativeElement, 'keyup');
    logger$.subscribe(evt => this.keys += evt.key);
  }
}
```

Notice that we get access to the native HTML input element through the `nativeElement` property of the template reference variable. The result of using the `@ViewChild` decorator is an `ElementRef` object, which is a wrapper over the actual HTML element.

7. Open the `app.component.html` file and replace its content with the following HTML template:

```
<span>{{title}} app is running!</span>
<div>
  <app-key-logger></app-key-logger>
</div>
```

Run the application using the `ng serve` command and start pressing keys to verify the use of the key logger that we have just created:

Figure 7.2: Key logger output

An essential aspect of observables is using operators and chaining observables together, enabling **rich composition**. Observable operators look like array methods when we want to use them. For example, a `map` operator for observables is used similarly to the `map` method of an array. In the following section, we will learn about the **RxJS library**, which provides these operators, and learn about some of them through examples.

The RxJS library

As mentioned previously, Angular comes with a peer dependency on RxJS, the JavaScript flavor of the ReactiveX library that allows us to create observables out of a large variety of scenarios, including the following:

- Interaction events
- Promises
- Callback functions
- Events

In this sense, reactive programming does not aim to replace asynchronous patterns, such as promises or callbacks. All the way around, it can leverage them as well to create observable sequences.

RxJS has built-in support for a wide range of composable operators to transform, filter, and combine the resulting event streams. Its API provides convenient methods for observers to subscribe to these streams so that our components can respond accordingly to state changes or input interaction. Let’s see some of these operators in action in the following sections.

Creating observables

We have already learned how to create an observable from a DOM event using the `fromEvent` operator. Two other popular operators concerned with observable creation are the `of` and `from` operators.

The `of` operator is used to create an observable from values such as numbers:

```
import { of } from 'rxjs';
const values = of(1, 2, 3);
values.subscribe(value => console.log(value));
```

The previous snippet will print the numbers `1`, `2`, and `3` in the console window *in sequence*.

The `from` operator is used to convert an array or a promise to an observable:

```
import { from } from 'rxjs';
const values = from([1, 2, 3]);
values.subscribe(value => console.log(value));
```

The `from` operator is also very useful when converting promises or callbacks to observables. We could wrap the `onComplete` method in the `constructor` of the `AppComponent` class as follows:

```
constructor() {
  const complete$ = from(this.onComplete());
  complete$.subscribe(this.setTitle);
}
```

The `from` operator is an excellent way to start migrating from promises to observables in your Angular application if you have not done so already!

Besides creating observables, the RxJS library also contains a couple of handy operators to manipulate and transform data emitted from observables.

Transforming observables

We have already learned how to create a numeric-only directive in *Chapter 5*, *Enrich Applications using Pipes and Directives*. We will now use RxJS operators to accomplish the same thing in our key logger component:

1. Open the `key-logger.component.ts` file and import the `tap` operator from the `rxjs` npm package:

```
import { fromEvent, tap } from 'rxjs';
```

2. Refactor the `ngOnInit` method as follows:

```
ngOnInit(): void {
  const logger$ = fromEvent<KeyboardEvent>(this.input?.nativeElement, 'keyup');
  logger$.pipe(
    tap(evt => this.keys += evt.key)
  ).subscribe();
}
```

The `pipe` operator is used to link and combine multiple operators separated by commas. We can think of it as a recipe that defines the operators that should be applied to an observable. One of them is the `tap` operator, which is used when we want to do something with the data emitted without modifying it.

3. We want to exclude non-numeric values that the `logger$` observable emits. We already get the actual key pressed from the `evt` property, but it returns alphanumeric values. It would not be efficient to list all non-numeric values and exclude them manually. Instead, we will use the `map` operator to get the actual Unicode value of the key. It behaves similarly to the `map` method of an array as it returns an observable with a modified version of the initial data. Import the `map` operator from the `rxjs` npm package:

```
import { fromEvent, map, tap } from 'rxjs';
```

4. Add the following snippet above the `tap` operator in the `ngOnInit` method:

```
map(evt => evt.key.charCodeAt(0)),
```

5. We can now add the `filter` operator, which operates similarly to the `filter` method of an array to exclude non-numeric values. Import the `filter` operator from the `rxjs` npm package:

```
import { filter, fromEvent, map, tap } from 'rxjs';
```

6. Add the following snippet after the `map` operator in the `ngOnInit` method:

```
filter(code => (code > 31 && (code < 48 || code > 57)) === false),
```

   In the preceding snippet, we omit the `return` statement from the arrow function. It is a shorthand syntax that requires writing the arrow function in one line without brackets.

7. The observable currently emits Unicode character codes. We need to convert them back to actual keyboard characters to display them on the HTML template. Refactor the `tap` operator to accommodate this change:

```
tap(digit => this.keys += String.fromCharCode(digit))
```

As a final touch, we will add an input binding in the component to toggle the numeric-only feature on and off, conditionally:

1. Add the `Input` artifact in the `import` statement of the `@angular/core` npm package:

```
import { Component, ElementRef, OnInit, ViewChild, Input } from '@angular/core';
```

2. Add a `numeric` input property in the `KeyLoggerComponent` class:

```
@Input() numeric = false;
```

3. Refactor the `filter` operator in the `ngOnInit` method so that it takes into account the `numeric` property:

```
filter(code => {
  if (this.numeric) {
    return (code > 31 && (code < 48 || code > 57)) === false;
  }
  return true;
})
```

The `logger$` observable will filter non-numeric values only if the `numeric` input property is `true`.

The `ngOnInit` method should finally look like the following:

```
ngOnInit(): void {
  const logger$ = fromEvent<KeyboardEvent>(this.input?.nativeElement, 'keyup');
  logger$.pipe(
    map(evt => evt.key.charCodeAt(0)),
    filter(code => {
```

```
      if (this.numeric) {
        return (code > 31 && (code < 48 || code > 57)) === false;
      }
      return true;
    }),
    tap(digit => this.keys += String.fromCharCode(digit))
  ).subscribe();
}
```

We have already seen RxJS operators manipulating observables that return primitive data types such as numbers, strings, and arrays. However, there are additional operators that we can use to work with observables that also return observables as values.

Higher-order observables

Observables that operate on other observables *of* observables are called **higher-order observables**. Higher-order observables have an *inner* observable that contains the actual values we are interested in using. We can use specific RxJS operators to *flatten* the inner observable and extract its values. The most used flattened operators in Angular development are the `switchMap` and `mergeMap` operators.

The `switchMap` operator takes an observable as a source and applies a given function to each item, returning an inner observable for each one. The operator returns an output observable with values emitted from each inner

observable. As soon as an inner observable emits a new value, the output observable stops receiving values from the other inner observables.

We will now investigate how `switchMap` is usually used in Angular applications. Remember the `ProductsService` and `ProductViewService` classes in the previous chapter? We will now convert them to use observables instead of plain arrays and learn how to combine them using the `switchMap` operator.

If you want to follow along with the source code of this chapter, make sure that you copy the `products` folder from the source code of *Chapter 6*, *Managing Complex Tasks with Services*, in the `src\app` folder of your current project. Otherwise, you can continue from where you left off with your project in the previous chapter.

In this chapter, we will work with the product list and product view components for simplicity and repeatability. However, the `products` module also contains other components and services that must be modified to reflect the use of observables in the codebase. We encourage you to make these changes yourselves. The source code described in the *Technical requirements* section has been trimmed down to compile successfully without those components and services.

Let's get started:

1. Open the `products.service.ts` file and import the `of` and `Observable` artifacts from the `rxjs` npm package:

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
```

```
import { Product } from './product';
```

2. Extract the `products` array into a separate service property to enhance code readability:

```
private products = [
  {
    name: 'Webcam',
    price: 100
  },
  {
    name:  'Microphone',
    price: 200
  },
  {
    name: 'Wireless keyboard',
    price: 85
  }
];
```

3. Modify the `getProducts` method so that it returns the `products` property:

```
getProducts(): Observable<Product[]> {
  return of(this.products);
}
```

In the preceding snippet, we use the `of` operator to create a new observable from the `products` array.

4. Open the `product-view.service.ts` file and add the following `import` statement:

```
import { Observable, of, switchMap } from 'rxjs';
```

5. Modify the `getProduct` method to return an observable of a `Product` object:

```
getProduct(id: number): Observable<Product> {
  return this.productService.getProducts().pipe(
    switchMap(products => {
      if (!this.product) {
        this.product = products[id];
      }
      return of(this.product);
    })
  );
}
```

In the preceding method, there are a lot of RxJS mechanics involved. We call the `getProducts` method of the `ProductsService` class, which returns an observable of products. We also use the `pipe` operator to chain the observable of products with the observable returned from the `switchMap` operator. The `switchMap` operator creates a new inner observable for each product emitted from the source observable, using the `of` operator. Finally, the `getProduct` method returns the output observable that results from the `pipe` operator.

As the name implies, the `switchMap` operator cancels any current inner observable that is active and *switches* to a new one when the source observable emits a new value. If we would like to wait for all inner observables to complete, we could use the `mergeMap` operator from RxJS. The `mergeMap` operator, as the name implies, *merges* values from all inner observables into one. The only change we must make to start using the `mergeMap` operator is to modify the `product-view.service.ts` accordingly:

```
import { Injectable } from '@angular/core';
import { Observable, of, mergeMap } from 'rxjs';
import { ProductsService } from '../products.service';
import { Product } from '../product';
@Injectable()
export class ProductViewService {
  private product: Product | undefined;
  constructor(private productService: ProductsService) { }
  getProduct(id: number): Observable<Product> {
    return this.productService.getProducts().pipe(
      mergeMap(products => {
        if (!this.product) {
          this.product = products[id];
        }
        return of(this.product);
      })
    );
  }
```

```
}
```

We have started using observables in our Angular services using the RxJS library. However, our application is now broken because our Angular components did not reflect that change. We need to adjust them so they can interact with the new observable-based services and get data using observable streams.

Subscribing to observables

We have already learned that an observer needs to subscribe to an observable in order to start getting emitted data. Our products and product-view services currently emit product data using observables. We must modify their respective components to subscribe and get these data:

1. Open the `product-list.component.ts` file and create a `getProducts` method in the `ProductListComponent` class:

```
private getProducts() {
  this.productService.getProducts().subscribe(products => {
    this.products = products;
  });
```

```
}
```

In the preceding method, we subscribe to the `getProducts` method of the `ProductsService` class because it returns an observable instead of a plain `products` array. The `products` array is returned inside the `subscribe` method, where we set the `products` component property to the array emitted from the observable.

2. Modify the `ngOnInit` method so that it calls the newly created `getProducts` method:

```
ngOnInit(): void {
  this.getProducts();
}
```

We could have added the body of the `getProducts` method inside the `ngOnInit` method directly, but we did not. Component life cycle event methods should be as clear and concise as possible. Always try to extract their logic in a separate method for clarity.

3. Open the `product-view.component.ts` file and create a `getProduct` method in the `ProductViewComponent` class:

```
private getProduct() {
  this.productviewService.getProduct(this.id).subscribe(product => {
```

```
    if (product) {
      this.name = product.name;
    }
  });
}
```

Similarly to step 1, we subscribe to the `getProduct` method of the `ProductViewService` class and set the `name` component property inside the `subscribe` method.

4. We also need to modify the `ngOnInit` method so that it calls the `getProduct` method:

```
ngOnInit(): void {
  this.getProduct();
}
```

If you are working with the Angular project you created in the previous chapter, skip steps 5 and 6.

5. Open the `app.module.ts` file and import `ProductsModule`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { KeyLoggerComponent } from './key-logger/key-logger.component';
```

```
import { ProductsModule } from './products/products.module';
@NgModule({
  declarations: [
    AppComponent,
    KeyLoggerComponent
  ],
  imports: [
    BrowserModule,
    ProductsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

6. Add the product list component in the `app.component.html` file:

```
<span>{{title}} app is running!</span>
<div>
  <app-key-logger></app-key-logger>
</div>
<app-product-list></app-product-list>
```

Run the application using the `ng serve` command, and you should see the product list displayed on the page successfully:

Figure 7.3: Product list

As depicted in the previous image, we have achieved the same result of displaying the product list as in *Chapter 6*, *Managing Complex Tasks with Services*, but using observables. It may not be evident at once, but we have set the foundation for working with the Angular HTTP client that is based on observables. In *Chapter 8*, *Communicating with Data Services over HTTP*, we will explore the HTTP client in more detail.

When we subscribe to observables, we are prone to potential memory leaks if we do not clean them up on time. In the following section, we will learn about different ways to accomplish that.

Unsubscribing from observables

When we subscribe to an observable, we create an observer that listens for changes in a data stream. The observer watches the stream continuously while the subscription remains active. When a subscription is active, it reserves memory in the browser and consumes certain resources. If we do not tell the observer to unsubscribe at some point and clean up any resources, the subscription to the observable will *possibly* lead to a memory leak.

An observer usually needs to unsubscribe when the Angular component that has created the subscription needs to be destroyed.

Some of the most well-known techniques to use when we are concerned with unsubscribing from observables are the following:

- Unsubscribe from an observable manually.
- Use the `async` pipe in a component template.

Let’s see both techniques in action in the following sections.

Destroying a component

A component has life cycle events we can use to hook on and perform custom logic, as we learned in *Chapter 4*, *Enabling User Experience with Components*. One of them is the `ngOnDestroy` event, which is called when the component is destroyed and no longer exists.

Recall `ProductListComponent` and `ProductViewComponent`, which we used earlier in our examples. They subscribe to the appropriate methods of `ProductsService` and `ProductViewService` upon component initialization. When components are destroyed, the reference of the subscriptions stays active, which may lead to unpredictable behavior. We need to manually unsubscribe when components are destroyed to clean up any resources properly:

1. Open the `product-list.component.ts` file and add the following `import` statement:

```
import { Subscription } from 'rxjs';
```

2. Create the following property in the `ProductListComponent` class:

```
private productsSub: Subscription | undefined;
```

3. Assign the `productsSub` property to the subscription in the `getProducts` method:

```
private getProducts() {
  this.productsSub = this.productService.getProducts().subscribe(products => {
```

```
    this.products = products;
  });
}
```

4. Import the `OnDestroy` lifecycle hook from the `@angular/core` npm package:

```
import { AfterViewInit, Component, OnDestroy, OnInit, ViewChild } from '@angular/core';
```

5. Add `OnDestroy` to the implemented interface list of the `ProductListComponent` class.
6. Implement the `ngOnDestroy` method as follows:

```
ngOnDestroy(): void {
  this.productsSub?.unsubscribe();
}
```

The `unsubscribe` method removes an observer from the active listeners of a subscription and cleans up any reserved resources.

That's a lot of boilerplate code to unsubscribe from a single subscription. It may quickly become unreadable and unmaintainable if we have many subscriptions. Can we do better than this? Yes, we can!

We can use a particular type of observable called `Subject`, which extends an `Observable` object as it is both an observer and an observable. It can emit values to multiple observers, whereas an `Observable` object unicasts only

to one observer at a time. We have already met such an object before in *Chapter 4*, *Enabling User Experience with Components*. The `EventEmitter` class from the `@angular/core` npm package that we used in the output binding of a component is a `Subject`. Other cases that a `Subject` can be used for are the following:

- To pass data between components using observables
- To implement a mechanism with *search as you type* features

We will explore the way of unsubscribing from observables using a `Subject` in the product view component:

1. Open the `product-view.component.ts` file and add the following `import` statement:

```
import { Subject, takeUntil } from 'rxjs';
```

   The `takeUntil` artifact is an RxJS operator that unsubscribes from an observable when it completes. The `Subject` artifact is also part of the `rxjs` npm package.

2. Create a `productSub` property in the `ProductViewComponent` class and initialize it with an instance of the `Subject` class:

```
private productSub = new Subject<void>();
```

3. Modify the `getProduct` method to use the `takeUntil` operator:

```
private getProduct() {
  this.productviewService.getProduct(this.id).pipe(
    takeUntil(this.productSub)
  ).subscribe(product => {
    if (product) {
      this.name = product.name;
    }
  });
}
```

In the preceding method, we use the `pipe` operator to chain the `takeUntil` operator with the subscription from the `getProduct` method of the `ProductViewService` class. The `takeUntil` operator accepts a parameter of the subscription that waits for completion.

4. Import the `OnDestroy` lifecycle hook from the `@angular/core` npm package:

```
import { Component, Input, OnDestroy, OnInit } from '@angular/core';
```

5. Add `OnDestroy` to the implemented interface list of the `ProductViewComponent` class.
6. Implement the `ngOnDestroy` method so that it completes the `productSub`:

```
ngOnDestroy(): void {
  this.productSub.next();
```

```
  this.productSub.complete();
}
```

Before completing a subject, we must call its `next` method to emit any last values to its subscribers.

That's it! We have now converted our subscription in a more declarative way that is more readable. But the problem of maintainability still exists. Our components are now unsubscribing from their observables manually. We can solve that using a special-purpose Angular pipe, the `async` pipe, which allows us to unsubscribe automatically with less code.

Using the async pipe

The `async` pipe is an Angular built-in pipe that is used in conjunction with observables, and its role is two-fold. It helps us to type less code, and it saves us from having to set up and tear down a subscription. It automatically subscribes to an observable and unsubscribes when the component is destroyed. We will use it to simplify the code of the product list component:

1. Open the `product-list.component.ts` file and import the `Observable` artifact from the `rxjs` npm package:

```
import { Subscription, Observable } from 'rxjs';
```

2. Convert the `products` component property to an observable:

```
products$: Observable<Product[]> | undefined;
```

3. Assign the `getProducts` method of the `ProductsService` class to the `products$` component property:

```
private getProducts() {
  this.products$ = this.productService.getProducts();
}
```

The body of the `getProducts` method has now been reduced to one line and has become more readable.

Feel free to remove any unused code related to the old `productsSub` property to improve the component furthermore.

4. Open the `product-list.component.html` file and modify the unordered list element to use the `async` pipe:

```
<h2>Product List</h2>
<ul>
  <li
    *ngFor="let product of (products$ | async)! | sort; let i=index"
    (click)="selectedProduct = product">
      <app-product-view [id]="i"></app-product-view>
  </li>
```

```
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [product]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
</ng-template>
```

In the preceding template, we use the **non-null assertion operator** to type-cast the result of passing the `products$` observable through the `async` pipe. The non-null assertion operator tricks the TypeScript compiler into ignoring `null` and `undefined` values in template expressions. In strict mode, the `async` pipe assumes that the observable passed through it *may* be `null`, which comes against our declaration of the `products$` property.

That's it! We do not need to subscribe or unsubscribe from the observable manually anymore! The `async` pipe takes care of everything for us.

Summary

It takes much more than a single chapter to cover in detail all the great things we can do with the RxJS library. The good news is that we have covered all the tools and classes we need for basic Angular development. We learned what reactive programming is and how it can be used in Angular. We saw how to apply reactive techniques such as observables to interact with data streams. We explored the RxJS library and how we can use some of its operators to manipulate observables. We learned different ways of subscribing and unsubscribing from observables in Angular components.

The rest is just left to your imagination, so feel free to go the extra mile and put all of this knowledge into practice in your Angular applications. The possibilities are endless, and you have assorted strategies to choose from, ranging from promises to observables. You can leverage the incredible functionalities of the reactive operators and build amazing reactive experiences for your Angular applications.

As we have already highlighted, the sky's the limit. However, we still have a long and exciting road ahead. Now that we know how to consume asynchronous data in our components, let's discover how we can benefit from the power of the RxJS library when we want to communicate over HTTP. In the next chapter, we will learn how to use the Angular built-in HTTP client and consume data from a remote endpoint.

End of Chapter 7