Communicating with Data Services over HTTP

A real-world scenario for enterprise Angular applications is to connect to remote services and APIs to exchange data. The built-in Angular HTTP client provides out-of-the-box support for communicating with services over HTTP. The interaction of an Angular application with the HTTP client is based on RxJS observable streams, giving developers a rich set of capabilities for data access.

There are many possibilities to describe what you can do to connect to APIs through HTTP. In this book, we will only scratch the surface. Still, the insights covered in this chapter will give you all you need to connect your Angular applications to HTTP services in no time, leaving all you can do with them up to your creativity.

In this chapter, we will explore the following concepts:

- Communicating data over HTTP

- Introducing the Angular HTTP client
- Setting up a backend API
- Handling CRUD data in Angular
- Authentication and authorization with HTTP

Technical requirements

The chapter contains various code samples to walk you through the concept of the Angular HTTP client. You can find the related source code in the `ch08` folder of the following GitHub repository:

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

Communicating data over HTTP

Before we dive deeper into describing the Angular built-in HTTP client and how to use it to communicate with servers, let's talk about native HTTP implementations first. Currently, if we want to communicate with a server over HTTP using JavaScript, we can use the **fetch API**. It contains all the necessary methods to connect with a server and start exchanging data. You can see an example of how to fetch data in the following code:

```
fetch(url)
  .then(response => {
    return response.ok ? response.text() : '';
  })
  .then(result => {
    if (result) {
      console.log(result);
    } else {
      console.error('An error has occured');
    }
  });
```

Although the fetch API is promise-based, the promise that returns is not rejected in case of error. Instead, the request is unsuccessful when the `ok` property is not present in the `response` object. If the request to the remote URL completes, we can use the `text()` method of the `response` object to return the response text inside a new promise. Finally, in the second `then` callback, we display either the response text or a specific error message to the browser console.

To learn more details about the fetch API, check out the official documentation at https://developer.mozilla.org/docs/Web/API/fetch.

We have already learned that observables are more flexible for managing asynchronous operations. You are probably wondering how we can apply this pattern to an asynchronous scenario, such as consuming information from an HTTP service. You have so far become used to submitting asynchronous requests to AJAX services and then passing the response to a callback or a promise. Now, we will handle the call by returning an observable. The observable will emit the server response as an event in the context of a stream, which can be funneled through RxJS operators to digest the response better.

Let's try to convert the previous example with the fetch API to an observable. We use the `Observable` class constructor that we learned to wrap the `fetch` call in an observable stream. We replace the `log` method of the console with the appropriate `observer` object methods:

```
const request$ = new Observable(observer => {
  fetch(url)
```

```
    .then(response => {
      return response.ok ? response.text() : '';
    })
    .then(result => {
      if (result) {
        observer.next(result);
        observer.complete();
      } else {
        observer.error('An error has occured');
      }
    });
});
```

In the preceding snippet, the `next` method emits the response data back to subscribers as soon as they arrive. The `complete` method notifies them that no other data will be available in the stream. In the case of an error, the `error` method alerts subscribers that an error has occurred.

That's it! You have now built your custom HTTP client. Of course, this isn't much. Our custom HTTP client only handles a `GET` operation to get data from a remote endpoint. We are not handling many other operations of the HTTP protocol, such as `POST`, `PUT`, and `DELETE`. It was, however, essential to realize all the heavy lifting the HTTP client in Angular is doing for us. Another important lesson is how easy it is to take any asynchronous API and turn it into an observable one that fits nicely with the rest of our asynchronous concepts. So, let’s continue with Angular’s implementation of an HTTP service.

Introducing the Angular HTTP client

The built-in HTTP client of the Angular framework is a separate Angular library that resides in the `@angular/common` npm package under the `http` namespace. The Angular CLI installs this package by default when creating a new Angular project. To start using it, we need to import `HttpClientModule` in the main application module, `app.module.ts`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The `HttpClientModule` class provides various Angular services we can use to handle asynchronous HTTP communication. The most basic is the `HttpClient` service, which provides a robust API and abstracts all operations required to handle asynchronous connections through various HTTP methods. Its implementation was considered carefully to ensure that developers feel at ease while developing solutions that take advantage of this class.

In a nutshell, instances of the `HttpClient` service have access to various methods to perform common HTTP request operations, such as `GET`, `POST`, `PUT`, and every existing HTTP verb. In this book, we are interested in the most basic ones:

- `get`: This performs a `GET` operation to fetch data.

- `post`: This performs a `POST` operation to add new data.
- `put/patch`: This performs a `PUT/PATCH` operation to update existing data.
- `delete`: This performs a `DELETE` operation to remove existing data.

The previous HTTP operations constitute the primary operations for **Create Read Update Delete (CRUD)** applications. All the previous methods of the Angular HTTP client return an observable data stream. Angular components can use the RxJS library to subscribe to those methods and interact with a remote API. In the following section, we will explore how to use these methods and communicate with a remote API.

Setting up a backend API

A web CRUD application usually connects to a server and uses an HTTP backend API to perform operations on data. It fetches existing data, updates it, creates new data, or deletes it. In a real-world scenario, you will most likely interact with a real backend API service through HTTP. There are cases, though, where we do not have access to a real backend API:

- We may work remotely, and the server is only accessible through a VPN connection to which we do not have access.
- We want to set up a quick prototype for demo purposes.
- Available HTTP endpoints are not yet ready for consumption from the backend development team, a common problem when working in a large team of different types of developers.

To overcome all the previous obstacles during development, we can use a fake server such as the **Fake Store API**. The Fake Store API is a backend REST API available online that you can use when you need fake data for an e-commerce or e-shop web application. It can manage products, shopping carts, and users that are available in the JSON format. It exposes the following main endpoints:

- **products**: This manages a set of product items.
- **cart**: This manages the shopping cart of a user.
- **user**: This manages a collection of application users.
- **login**: This handles user authentication.

All operations that modify data in the previous endpoints do not persist data in the database. However, they return an indication if the operation was successful. All `GET` operations in the previous endpoints return a predefined collection of data.

In this chapter, we will work only with the products and login endpoints. However, we will revisit the `cart` endpoint later in the book.

You can read more about the service at https://fakestoreapi.com.

Handling CRUD data in Angular

CRUD applications are widespread in the Angular world, particularly in the enterprise sector. You will hardly find any web application that does not follow this pattern. Angular does a great job supporting this type of application by providing the `HttpClient` service. In this section, we will explore the Angular HTTP client by interacting with the `products` endpoint of the Fake Store API. In particular, we will create an Angular CRUD application to manage `products` from our products module in *Chapter 7*, *Being Reactive Using Observables and RxJS*. Let's get started by scaffolding our application:

1. Create a new Angular application using the following Angular CLI command:

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

2. Copy the global CSS styles from the code samples of *Chapter 6*, *Managing Complex Tasks with Services*, inside the `styles.css` file of the current Angular project.
3. Copy the `products` folder from *Chapter 7*, *Being Reactive using Observables and RxJS*, in the `src\app` folder of the current Angular project.
4. Open the `product-list.component.ts` file and remove the `providers` property from the `@Component` decorator. The application root injector already provides the `ProductsService` class.
5. Open the `product-list.component.html` file and modify the unordered list element so that it does not use the `<app-product-view>` component:

```
<h2>Product List</h2>
<ul>
  <li *ngFor="let product of (products$ | async)! | sort" (click)="selectedProduct = product">
    {{product.name}}
  </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>
```

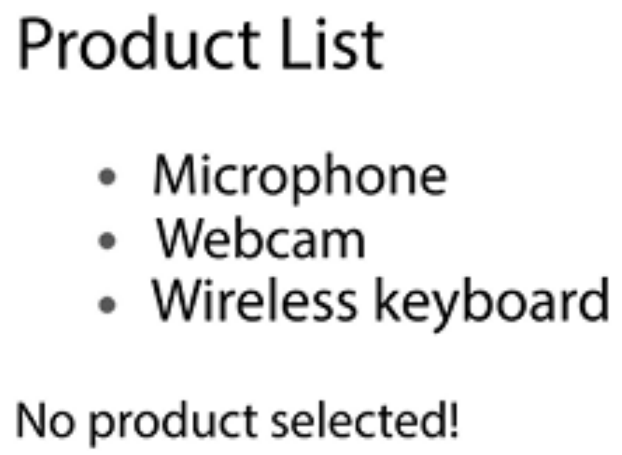
6. Open the `app.module.ts` file and import the `products` module:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { ProductsModule } from './products/products.module';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ProductsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

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

```
<app-product-list></app-product-list>
```

If we run the application using the `ng serve` command, we should see the product list displayed on the page correctly:



Product List

- Microphone
- Webcam
- Wireless keyboard

No product selected!

Figure 8.1: Product list

As we can see from the previous image, our application still displays static data. We need to modify it so that it gets product data from the Fake Store API.

Fetching data through HTTP

The product list component uses the `products` service to fetch and display product data. Data are currently hardcoded into the `products` property of the `ProductsService` class. In this section, we will modify our Angular application to work with live data from the Fake Store API:

1. Open the `app.module.ts` file and import the Angular module of the HTTP client:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
import { ProductsModule } from './products/products.module';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    ProductsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

2. Now, open the `products.service.ts` file and import the `HttpClient` class and the `map` RxJS operator:

```
import { Injectable } from '@angular/core';
import { map, Observable, of } from 'rxjs';
import { HttpClient } from '@angular/common/http';
import { Product } from './product';
```

3. Create the following interface after the `import` statements:

```
interface ProductDTO {
  title: string;
  price: number;
}
```

The preceding interface conforms to the product object model of the Fake Store API. It will act as an intermediate for transforming the response object from the API into our `Product` interface. The `Product` interface defined in the `product.ts` file contains a `name` property instead of a `title` that the

API requires. The suffix `DTO` comes from **Data Transfer Object (DTO)** and is often used to indicate that the interface is part of the backend API specification model.

Collaborating with the backend team that develops the API early in a project and agreeing on the structure of the models you exchange will avoid creating intermediate interfaces.

4. Create the following property in the `ProductsService` class that represents the API products endpoint:

```
private productsUrl = 'https://fakestoreapi.com/products';
```

5. Inject `HttpClient` in the `constructor` of the `ProductsService` class:

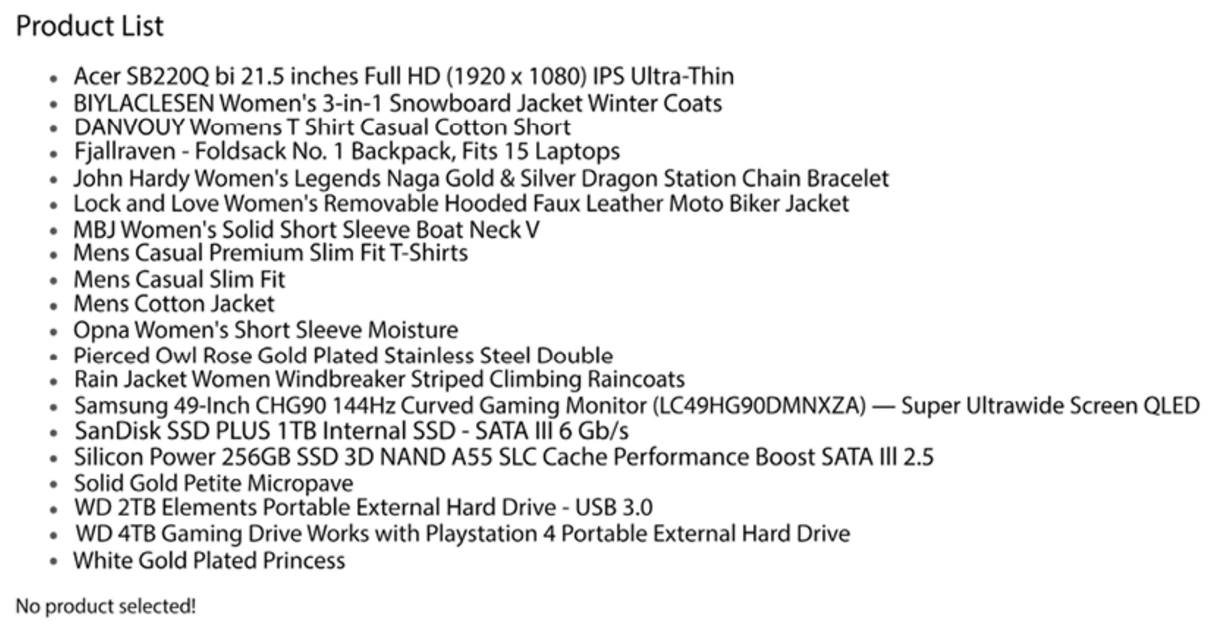
```
constructor(private http: HttpClient) { }
```

6. Modify the `getProducts` method so that it uses the `HttpClient` service to get the list of products:

```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return {
        name: product.title,
        price: product.price
      }
    }))
  );
}
```

In the preceding method, we use the `get` method of the `HttpClient` class and pass the `products` endpoint of the API as a parameter. We also define the `ProductDTO[]` as a generic type in the `get` method to indicate that the response from the API contains a list of `ProductDTO` objects. To transform the response from the API into a list of `Product` objects that our components understand, we use the `map` operator of the RxJS library.

If we run the application using the `ng serve` command, we should see a list of products from the API similar to the following:



Product List

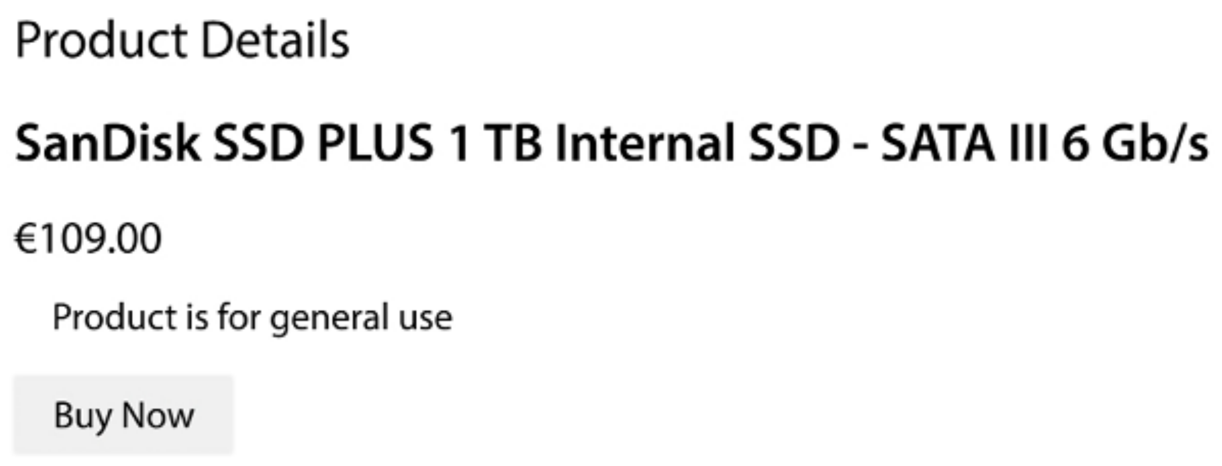
- Acer SB220Q bi 21.5 inches Full HD (1920 x 1080) IPS Ultra-Thin
- BIYLACLESEN Women's 3-in-1 Snowboard Jacket Winter Coats
- DANVOUY Womens T Shirt Casual Cotton Short
- Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops
- John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet
- Lock and Love Women's Removable Hooded Faux Leather Moto Biker Jacket
- MBJ Women's Solid Short Sleeve Boat Neck V
- Mens Casual Premium Slim Fit T-Shirts
- Mens Casual Slim Fit
- Mens Cotton Jacket
- Opna Women's Short Sleeve Moisture
- Pierced Owl Rose Gold Plated Stainless Steel Double
- Rain Jacket Women Windbreaker Striped Climbing Raincoats
- Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) — Super Ultrawide Screen QLED
- SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s
- Silicon Power 256GB SSD 3D NAND A55 SLC Cache Performance Boost SATA III 2.5
- Solid Gold Petite Micropave
- WD 2TB Elements Portable External Hard Drive - USB 3.0
- WD 4TB Gaming Drive Works with Playstation 4 Portable External Hard Drive
- White Gold Plated Princess

No product selected!

Figure 8.2: Product list from the Fake Store API

You can now remove the `products` property of the `ProductsService` class.

If you click on a product from the list, you will notice that the product details are shown correctly:


Product Details

SanDisk SSD PLUS 1 TB Internal SSD - SATA III 6 Gb/s

€109.00

Product is for general use

Figure 8.3: Product details

The product details component continues to work as expected because we pass the selected product as an input property from the product list:

```
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [product]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
```

We will change the previous behavior and get the product details directly from the API using an HTTP GET request:

1. The Fake Store API contains an endpoint method that we can use to get details for a specific product based on its ID. We do not currently have an ID property in our product model, so first, let's add one in the `product.ts` file:

   ```
   export interface Product {
     id: number;
     name: string;
     price: number;
   }
   ```

2. Open the `products.service.ts` file and modify the `ProductDTO` interface to include the `id` property:

   ```
   interface ProductDTO {
     id: number;
     title: string;
     price: number;
   }
   ```

3. Create the following helper method to transform a `ProductDTO` object into a `Product` object because we will need it often inside the service:

```
private convertToProduct(product: ProductDTO): Product {
  return {
    id: product.id,
    name: product.title,
    price: product.price
  };
}
```

4. Refactor the `getProducts` method to use the newly created `convertToProduct` helper method:

```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    }))
  );
}
```

5. Create a new `getProduct` method that accepts the product `id` as a parameter and initiates a GET request to the API based on that `id`:

```
getProduct(id: number): Observable<Product> {
  return this.http.get<ProductDTO>(`${this.productsUrl}/${id}`).pipe(
    map(product => this.convertToProduct(product))
  )
}
```

The preceding method uses the `get` method of the `HttpClient` service. It accepts the `products` endpoint followed by the product `id` as a parameter. It also defines the `ProductDTO` interface as generic because the result from the backend API will be a `ProductDTO` object.

We need to adjust the product detail component to get product details through the previous method instead of the input binding. We will still use input binding for passing the product `id` from the product list to the product detail component:

1. Open the `product-detail.component.ts` file and import the `Observable` and `ProductsService` artifacts:

```
import { Observable } from 'rxjs';
import { ProductsService } from '../products.service';
```

2. Create the following properties in the `ProductDetailComponent` class:

```
@Input() id = -1;
product$: Observable<Product> | undefined;
```

The `id` component property will be used to pass the `id` of the selected product from the list. The `product$` property will be used to call the `getProduct` method from the service.

3. Inject `ProductsService` in the `constructor` of the `ProductDetailComponent` class:

```
constructor(private productService: ProductsService) { }
```

4. Modify the `ngOnChanges` method as follows:

```
ngOnChanges(): void {
  this.product$ = this.productService.getProduct(this.id);
}
```

   We assign the value of the `getProduct` method from `ProductsService` to the `product$` component property every time a new `id` is passed using the input binding. We do not want to subscribe to the `getProduct` observable because we will use the `async` pipe in the component template that will do it for us automatically.

5. Open the `product-detail.component.html` file and replace its content with the following HTML template:

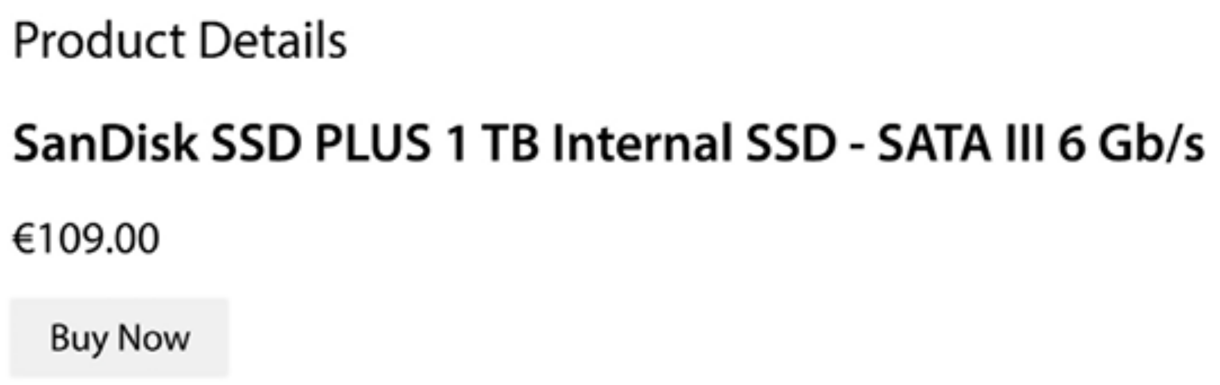
```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <p>
    <button (click)="buy()">Buy Now</button>
  </p>
</div>
```

In the preceding template, we have removed the `NgSwitch` directive and used the `async` pipe inside the `*ngIf` directive. The `async` pipe subscribes to the `product$` observable and saves the result in the `product` template variable.

6. Finally, open the `product-list.component.html` and bind the `id` of the `selectedProduct` property to the `id` input binding of the `app-product-detail` component:

```
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [id]="selectedProduct.id"
  (bought)="onBuy()">
</app-product-detail>
```

If we run the application using the `ng serve` command and select a product from the list, we will get an output like the following:


Product Details

SanDisk SSD PLUS 1 TB Internal SSD - SATA III 6 Gb/s

€109.00

Figure 8.4: Product details

We have already learned how to get a list of items and a single item from a backend API and covered the **Read** part of a CRUD operation. In the following section, we cover the remaining ones mainly concerned with modifying data.

Modifying data through HTTP

Modifying data in a CRUD application usually refers to adding new data and updating or deleting existing data. To demonstrate how to implement such functionality in an Angular application using the `HttpClient` service, we will make the following changes to our application:

- Create an Angular component to add new products.
- Modify the product detail component to change the price of an existing product.
- Add a button in the product detail component for deleting an existing product.

We will start with the component for adding new products.

Adding new products

To add a new product through our application, we need to send the name and price of a new product into the Fake Store API. Before implementing the required functionality for adding products, we must refactor the

`ProductListComponent` class in the `product-list.component.ts` file as follows:

```
export class ProductListComponent implements OnInit {
  selectedProduct: Product | undefined;
  products: Product[] = [];
  constructor(private productService: ProductsService) {}
  ngOnInit(): void {
    this.getProducts();
  }
  onBuy() {
    window.alert(`You just bought ${this.selectedProduct?.name}!`);
  }
  private getProducts() {
    this.productService.getProducts().subscribe(products => {
      this.products = products;
    });
  }
}
```

You may ask yourself why we are doing this. The `async` pipe is best suited for read-only data. In this case, we want to modify the product list by adding a new product or removing an existing one later. Plain arrays are best suited in this scenario because we can easily manipulate them using standard array functions.

You may also notice that we no longer unsubscribe from the `getProducts` observable when the component is destroyed. The Angular built-in HTTP client does a great job of unsubscribing automatically in *most* cases. However, it is advisable to always unsubscribe from observables in your components.

We must also change the `product-list.component.html` file so that the unordered list element iterates over the `products` array:

```
<h2>Product List</h2>
<ul>
  <li *ngFor="let product of products | sort" (click)="selectedProduct = product">
    {{product.name}}
  </li>
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [id]="selectedProduct.id"
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
```

```
</ng-template>
```

We can now start building the feature for adding new products to our application:

1. Open the `products.service.ts` file and add the following `addProduct` method:

```
addProduct(name: string, price: number): Observable<Product> {
  return this.http.post<ProductDTO>(this.productsUrl, {
    title: name,
    price: price
  }).pipe(
    map(product => this.convertToProduct(product))
  );
}
```

   In the preceding method, we use the `post` method of the `HttpClient` class and pass the `products` endpoint of the API, along with the details of a new product as parameters. The generic type defined in the `post` method indicates that the returned product from the API is a `ProductDTO` object. We transform the returned product into a `Product` type using the `convertToProduct` method.

2. Run the following Angular CLI command inside the `src\app\products` folder to create a new

component:

```
ng generate component product-create
```

3. Open the `product-create.component.ts` file and import the following artifacts:

```
import { Component, EventEmitter, Output } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
```

4. Add the following property in the `ProductCreateComponent` class using the `@Output` decorator:

```
@Output() added = new EventEmitter<Product>();
```

   We will use the preceding property to emit an event back to the product list component containing the new product we created.

5. Inject `ProductsService` into the `constructor` of the component class:

```
constructor(private productsService: ProductsService) {}
```

6. Add the following `createProduct` method in the component class:

```
createProduct(name: string, price: number) {
  this.productsService.addProduct(name, price).subscribe(product => {
    this.added.emit(product);
  });
}
```

The preceding method accepts the `name` and `price` of a product as parameters. It calls the `addProduct` method of the `ProductsService` class and then emits the newly created product using the `added` output event.

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

```
<div>
  <label for="name">Name</label>
  <input id="name" #name />
</div>
<div>
  <label for="price">Price</label>
  <input id="price" #price />
</div>
<div>
```

```
  <button (click)="createProduct(name.value, price.valueAsNumber)">Create</button>
</div>
```

In the preceding template, we bind the `createProduct` method to the `click` event of the `Create` button. We pass the value of each `<input>` element using the respective template reference variables, `name`, and `price`. The value of the `price` variable is converted to a number using the `valueAsNumber` property because it is in string format by default.

8. Open the `styles.css` file and add the following CSS styles to give a nice look and feel to our new component:

```
input {
  font-size: 14px;
  border-radius: 4px;
  padding: 8px;
  margin-bottom: 16px;
  border: 1px solid #BDBDBD;
}
label {
  font-size: 12px;
  font-weight: bold;
  margin-bottom: 4px;
  display: block;
}
```

We have already created the component for adding new products. All we have to do now is connect it with the product list component so that when a new product is created, it is also added to the product list:

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

   ```
   onAdd(product: Product) {
     this.products.push(product);
   }
   ```

   The preceding method adds a new product to the list. It will be called when a new product is added through the product create component.

2. Open the `product-list.component.html` file and add the following HTML content:

   ```
   <app-product-create (added)="onAdd($event)"></app-product-create>
   ```

In the preceding snippet, we add the product create component for adding new products and bind the `onAdd` component method to the `added` event. The `$event` object represents the newly added product.

If we now run our Angular application using the `ng serve` command, we should see the component for adding new products at the end of the page:

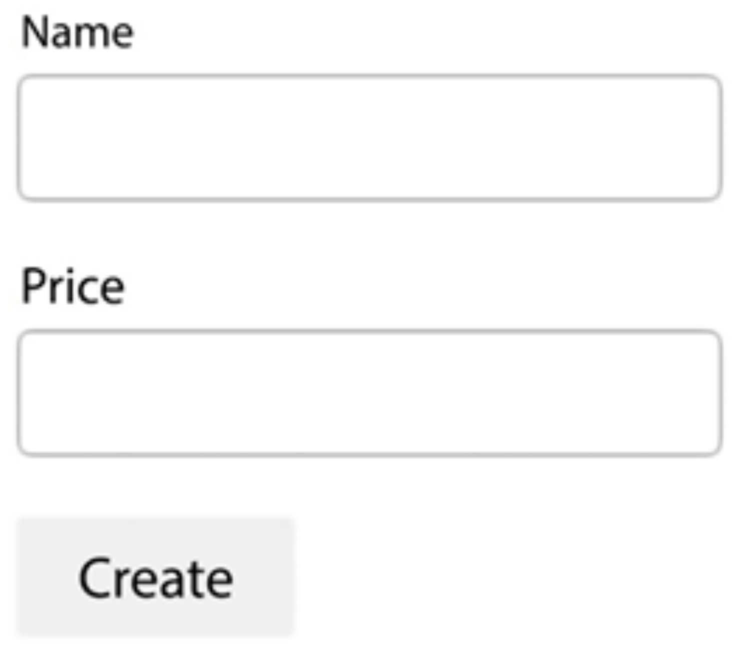


Figure 8.5: Create a product

To experiment, try to add a new product by filling in its details, clicking on the **Create** button, and verifying that the new product has been added to the list.

Do not try to select a new product from the list. Remember that any new data sent to the Fake Store API are not persisted in the database.

The next feature we will add to our CRUD application is to modify data by changing the price of an existing product.

Updating product price

The price of a product in an e-commerce application may need to change at some point. We need to provide a way for our users to update that price through our application:

1. Open the `products.service.ts` file and add a new method for updating a product:

```
updateProduct(id: number, price: number): Observable<void> {
  return this.http.patch<void>(`${this.productsUrl}/${id}`, { price });
}
```

In the preceding method, we use the `patch` method of the `HttpClient` service to send the details of the product that we want to modify to the API. Alternatively, we could use the `put` method of the HTTP client. The `patch` method should be used when we want to update only a subset of an object, whereas the `put` method interacts with all object properties. We do not want to update the product name in this case, so we use the `patch` method. Both methods accept the API endpoint and the object we want to update as parameters.

The return type of the `updateProduct` method is set to `Observable<void>` because we are not currently interested in the result of the HTTP request. We only need to know if it was successful or not.

2. Open the `product-detail.component.ts` file and add the following method:

```
changePrice(product: Product, price: number) {
  this.productService.updateProduct(product.id, price).subscribe(() => {
    alert(`The price of ${product.name} was changed!`);
  });
}
```

The preceding method accepts an existing product and its new price as parameters. It calls the `updateProduct` method of the `ProductsService` class and displays an alert message if the product is updated successfully.

3. Open the `product-detail.component.html` file and add an `<input>` and a `<button>` element after the `<span>` element:

```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <input placeholder="New price" #price/>
  <button (click)="changePrice(product, price.valueAsNumber)">Change</button>
  <p>
    <button (click)="buy()">Buy Now</button>
  </p>
</div>
```

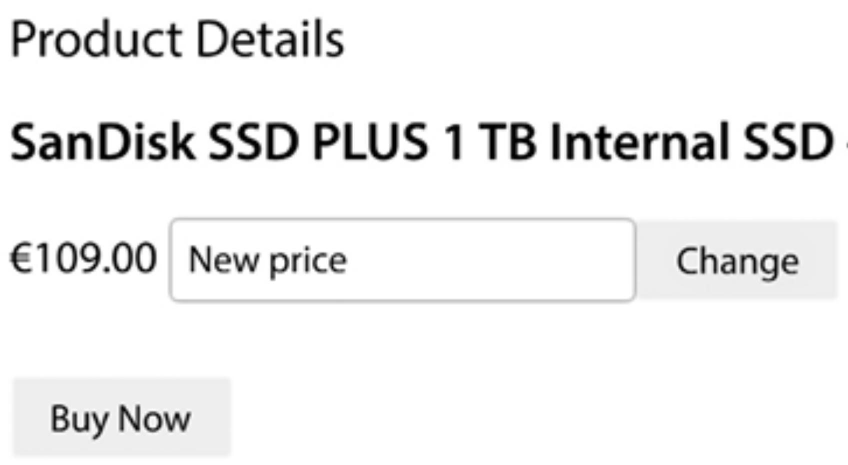
   The `<input>` element is used to enter the new price of the product and defines the `price` template reference variable. The `click` event of the `<button>` element is bound to the `changePrice` method that passes the current `product` object and the numeric value of the `price` variable.

4. Finally, open the `product-detail.component.css` file and add the following CSS styles:

```
input {
  margin-left: 5px;
}
```

The preceding styles will give space between the `<span>` element that displays the current price and the `<input>` element that accepts the new one.

5. Run the `ng serve` command to start the Angular application and select a product from the list. The product detail component should look like the following:



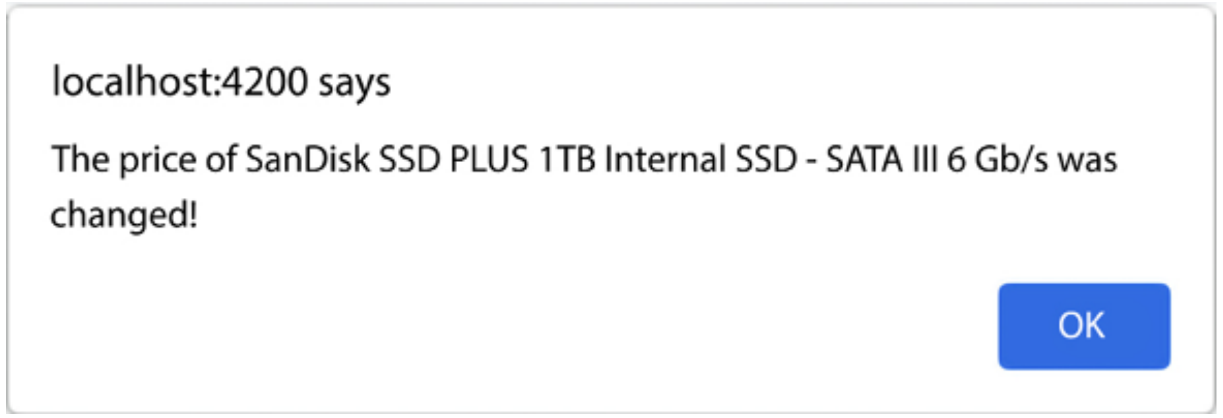
Product Details

SanDisk SSD PLUS 1 TB Internal SSD - SATA III 6 Gb/s

€109.00

Figure 8.6: Product details

6. Enter a price in the **New price** input box and click the **Change** button. You should see an alert dialog containing the product name and stating that the price has changed:



localhost:4200 says

The price of SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s was changed!

Figure 8.7: Change price alert

We can now modify a product by changing its price.

If you select the product again from the list, you will notice that the price has not been updated. Remember that any modifications to existing data sent to the Fake Store API are not persisted in the database.

The next and final step of our CRUD application will be to delete an existing product.

Removing a product

Deleting a product from an e-shop application is not very common. However, we need to provide the functionality in case users enter wrong or invalid data and want to delete them afterward. In our application, deleting an existing product will be available from the product details component:

1. Open the `products.service.ts` file and add the following method:

   ```
   deleteProduct(id: number): Observable<void> {
     return this.http.delete<void>(`${this.productsUrl}/${id}`);
   }
   ```

   The preceding method uses the `delete` method of the `HttpClient` service and passes the `products` endpoint, together with the product `id` we want to delete in the API.

   Similarly to the `updateProduct` method, we are not interested in the result of the API call. So, the signature of the method indicates that it returns `Observable<void>` as a type.

2. Open the `product-detail.component.ts` file and create a new output property in the `ProductDetailComponent` class:

```
@Output() deleted = new EventEmitter();
```

The preceding property will be used to notify the product list component that the selected product has been deleted. The product list component will then be responsible for removing the product from the list.

3. In the same component class, create the following method:

```
remove(product: Product) {
  this.productService.deleteProduct(product.id).subscribe(() => {
    this.deleted.emit();
  });
}
```

The preceding method calls the `deleteProduct` method of the `ProductsService` class and emits the `deleted` output event.

4. Open the `product-detail.component.html` file, create a `<button>` element, and bind its `click` event to the `remove` component method:

```
<div *ngIf="product$ | async as product">
```

```
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <input placeholder="New price" #price/>
  <button (click)="changePrice(product, price.valueAsNumber)">Change</button>
  <p>
    <button (click)="buy()">Buy Now</button>
    <button class="delete" (click)="remove(product)">Delete</button>
  </p>
</div>
```

5. Add an appropriate style for the new button in the `product-detail.component.css` file:

```
.delete {
  background-color: lightcoral;
  color: white;
  margin-left: 5px;
}
```

6. Open the `product-list.component.ts` file and add the following method:

```
onDelete() {
  this.products = this.products.filter(product => product !== this.selectedProduct);
  this.selectedProduct = undefined;
}
```

The preceding method will be called when the product detail component indicates that the selected product was deleted. It filters out the deleted product from the list and sets the `selectedProduct` to `undefined` so that there is no selected product on the page.

7. Open the `product-list.component.html` file and bind the `onDelete` component method to the `deleted` event of the `app-product-detail` component:

```
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [id]="selectedProduct.id"
  (deleted)="onDelete()"
  (bought)="onBuy()">
</app-product-detail>
```

If we run the application using the `ng serve` command and select a product from the list, we should see something like the following:

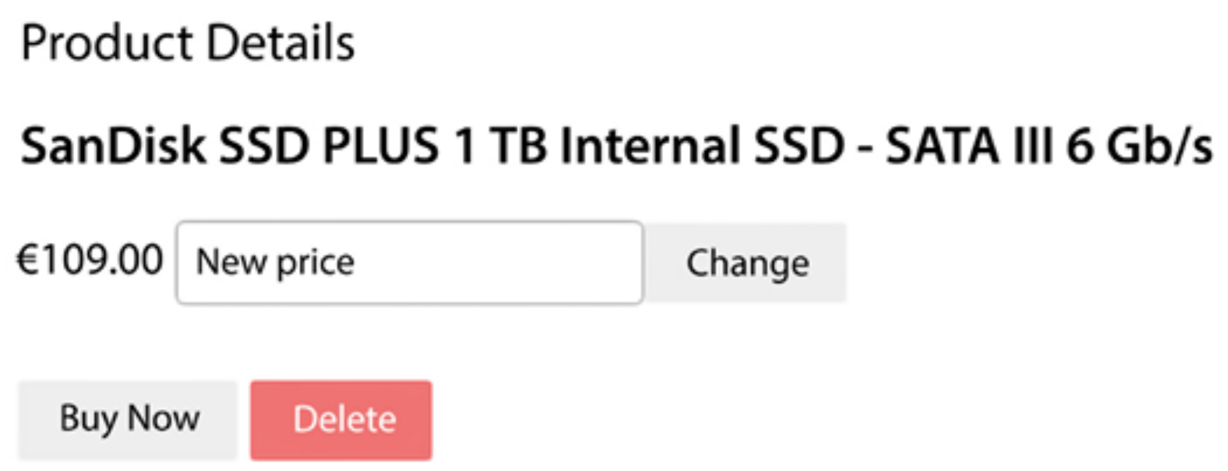


Figure 8.8: Product details

The product detail component now has a **Delete** button. When we click the button, the application deletes the product from the backend and hides it from the list.

If you reload your browser, you will notice that the product is still displayed on the list. Remember that any modifications to existing data sent to the Fake Store API are not persisted in the database.

The e-shop application we have built so far provides a **Buy Now** button that we can use to add a product to a cart. The button does not do much yet, but we will implement the full cart functionality in the following chapters. However, we should ensure that the feature will only be available to authenticated users.

In an Angular enterprise application, the product management feature must also be protected from unauthorized users. In this case, we would implement a more granular authorization scheme with user roles, where administrators would only be allowed to change and add products. We will not implement this feature, but we encourage you to experiment by yourselves.

The following section will teach us how to perform authentication and authorization in an Angular application.

Authentication and authorization with HTTP

The Fake Store API provides an endpoint for authenticating users with a username and a password. It contains a login method that accepts a username and a password as parameters and returns an authentication token. We will use the authentication token in our application to differentiate between a logged-in user and a guest.

The username and password are provided by a predefined pool of users at https://fakestoreapi.com/users.

We will explore the following authentication and authorization topics in this section:

- Authenticating with a backend API
- Authorizing users for certain features
- Authorizing HTTP requests using interceptors

Let’s get started with how to authenticate with the Fake Store API.

Authenticating with backend API

In Angular real-world applications, we usually consider authentication as an application feature. So, we will create a new Angular module that will handle authentication in our application. The module will contain an Angular component, allowing the user to log in to and log out of the application. An Angular service will be responsible for communicating with the Fake Store API and handling all authentication tasks.

Let’s get started by setting up the new Angular module:

1. Run the following command to create a new `auth` module:

```
ng generate module auth
```

2. Navigate inside the `src\app\auth` folder and run the following command to create a new Angular service:

```
ng generate service auth
```

3. Open the `auth.service.ts` file and add the following `import` statements:

```
import { HttpClient } from '@angular/common/http';
import { Observable, tap } from 'rxjs';
```

The `tap` artifact is an RxJS operator we use when we want to handle the emitted data from an observable without modifying it.

4. Create a `private` property for the authentication token in the `AuthService` class and inject the `HttpClient` service in the `constructor`:

```
export class AuthService {
  private token = '';
  constructor(private http: HttpClient) { }
}
```

We mark the property as `private` because we do not want sensitive data to be accessible outside the `AuthService` class.

5. Create a `login` method to allow users to log in to the Fake Store API:

```
login(): Observable<string> {
  return this.http.post<string>('https://fakestoreapi.com/auth/login', {
    username: 'david_r',
    password: '3478*#54'
  }).pipe(tap(token => this.token = token));
}
```

   The preceding method initiates a POST request to the API, using the `login` endpoint and passing predefined values for `username` and `password`. The observable returned from the POST request is passed to the `tap` operator, which sets the token received from the API to the `token` service property.

6. Create a `logout` method that resets the `token` property:

```
logout() {
  this.token = '';
}
```

We have already set up the business logic for authenticating users in our Angular application. In the following section, we will learn how to start using it and control authorization access in the application.

Authorizing user access

First, we will create an authentication component that will allow our users to log in to and log out of the application:

1. Run the following command inside the `src\app\auth` folder to create a new Angular component:

   ```
   ng generate component auth -export
   ```

   The preceding Angular CLI command will create the `auth` component and export it from the `auth` module so that other Angular modules can use it.

2. Open the `auth.component.ts` file and inject `AuthService`:

```
import { Component } from '@angular/core';
import { AuthService } from '../auth.service';
@Component({
  selector: 'app-auth',
  templateUrl: './auth.component.html',
  styleUrls: ['./auth.component.css']
})
export class AuthComponent {
  constructor(public authService: AuthService) { }
}
```

In the preceding scenario, we use the `public` access modifier to inject `AuthService` because we want it to be accessible from the component template.

Although we can use an Angular service directly in a component template, limiting its content inside the component class is considered a best practice. In this scenario, we follow the former for simplicity.

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

```
<button
  [hidden]="authService.isLoggedIn"
  (click)="authService.login().subscribe()"
```

```
>Login</button>
<button
  [hidden]="!authService.isLoggedIn"
  (click)="authService.logout()"
>Logout</button>
```

The preceding template contains two `<button>` elements for login/logout purposes. Each button is displayed conditionally according to the value of the `isLoggedIn` property of the `AuthService` class.

4. Open the `auth.service.ts` file and create the `isLoggedIn` getter property:

```
get isLoggedIn() { return this.token !== ''; }
```

According to the preceding property, a user is considered authenticated when a token is set in the application.

5. Open the `product-detail.component.ts` file and import the `AuthService` artifact:

```
import { AuthService } from '../../auth/auth.service';
```

6. Inject `AuthService` in the `constructor` of the `ProductDetailComponent`:

```
constructor(private productService: ProductsService, public authService: AuthService) { }
```

7. Open the `product-detail.component.html` file and use the `ngIf` directive to display the `Buy Now` button conditionally:

```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <input placeholder="New price" #price/>
  <button (click)="changePrice(product, price.valueAsNumber)">Change</button>
  <p>
    <button *ngIf="authService.isLoggedIn" (click)="buy()">Buy Now</button>
    <button class="delete" (click)="remove(product)">Delete</button>
  </p>
</div>
```

   In the preceding template, we used the `ngIf` directive, not the `hidden` attribute, because we want the button to be completely removed from the DOM.

8. Import the `auth` module into the main application module file, `app.module.ts`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
import { ProductsModule } from './products/products.module';
import { AuthModule } from './auth/auth.module';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    ProductsModule,
    AuthModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

9. Open the `app.component.html` file and add the `auth` component to the template:

```
<app-auth></app-auth>
<app-product-list></app-product-list>
```

Congratulations! You have gone a long way in adding basic authentication and authorization patterns to your Angular application.

It is common in enterprise applications to perform authorization in the business logic layer while communicating with the backend API. The backend API often requires certain method calls to pass the authentication token in each request through headers. We will learn how to work with HTTP headers in the following section.

Authorizing HTTP requests

The Fake Store API does not require authorization while communicating with its endpoints. However, consider that we are working with a backend API that expects all HTTP requests to contain an authentication token using HTTP **headers**. A common pattern in web applications is to include the token in an **Authorization** header. We can use HTTP headers in an Angular application by importing the `HttpHeaders` artifact from the `@angular/common/http` namespace and modifying our methods accordingly:

```
getProducts(): Observable<Product[]> {
  const options = {
    headers: new HttpHeaders({ Authorization: 'myAuthToken' })
  };
  return this.http.get<ProductDTO[]>(this.productsUrl, options).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    }))
  );
}
```

For simplicity, we are using a hardcoded value for the authentication token. In a real-world scenario, we may get it from the local storage of the browser or some other means.

All `HttpClient` methods accept an optional object as a parameter for passing additional options to an HTTP request. Request options can be an HTTP header, as in our case, or even query parameters. To set a header, we use the `headers` key of the `options` object and create a new instance of the `HttpHeaders` class as a value. The `HttpHeaders` object is a key-value pair that defines custom HTTP headers.

Now imagine what will happen if we need to pass the authentication token in all remaining methods of the

`ProductsService` class. We should go to each of them and write the same code repeatedly. Our code could quickly become cluttered and difficult to test. Luckily, the Angular built-in HTTP client has another feature we can use to help us in such a situation called **interceptors**.

An HTTP interceptor is an Angular service that intercepts HTTP requests and responses that pass through the Angular built-in HTTP client. It can be used in the following scenarios:

- When we want to pass custom HTTP headers in every request, such as an authentication token.
- When we want to display a loading indicator while we wait for a response from the server.
- When we want to provide a logging mechanism for every HTTP communication.

We can create an interceptor using the `generate` command of the Angular CLI. The following command will create an Angular interceptor named `auth`:

```
ng generate interceptor auth
```

Angular interceptors must be registered with an Angular module to use them. To register an interceptor with a module, import the `HTTP_INTERCEPTORS` injection token from the `@angular/common/http` namespace and add it to the `providers` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    ProductsModule,
    AuthModule
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
})
```

In the preceding snippet, we provide `AuthInterceptor` in the main application module, `AppModule`.

An HTTP interceptor must be provided in the same Angular module that imports `HttpClientModule`.

The provided object literal contains a key named `multi` that takes a boolean value. We set it to `true` to indicate that the `HTTP_INTERCEPTORS` injection token can accept multiple service instances. Using the `multi` option does not require the `providedIn` property in the `@Injectable` decorator of the service to be present. It also enables us to combine multiple interceptors, each satisfying a particular need. But how can they cooperate and play nicely altogether?

As we can see from the `auth.interceptor.ts` file, the interceptor is an Angular service that implements the `HttpInterceptor` interface:

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor() {}
  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    return next.handle(request);
  }
}
```

It implements the `intercept` method of the `HttpInterceptor` interface that accepts the following parameters:

- `request`: An `HttpRequest` object that indicates the current request
- `next`: An `HttpHandler` object that denotes the next interceptor in the chain

The purest form of an interceptor is to delegate requests to the next interceptor using the `handle` method. Thus, it is evident that the order in which we import interceptors in our Angular module matters. In the following diagram, you can see how interceptors process HTTP requests and responses according to their import order:

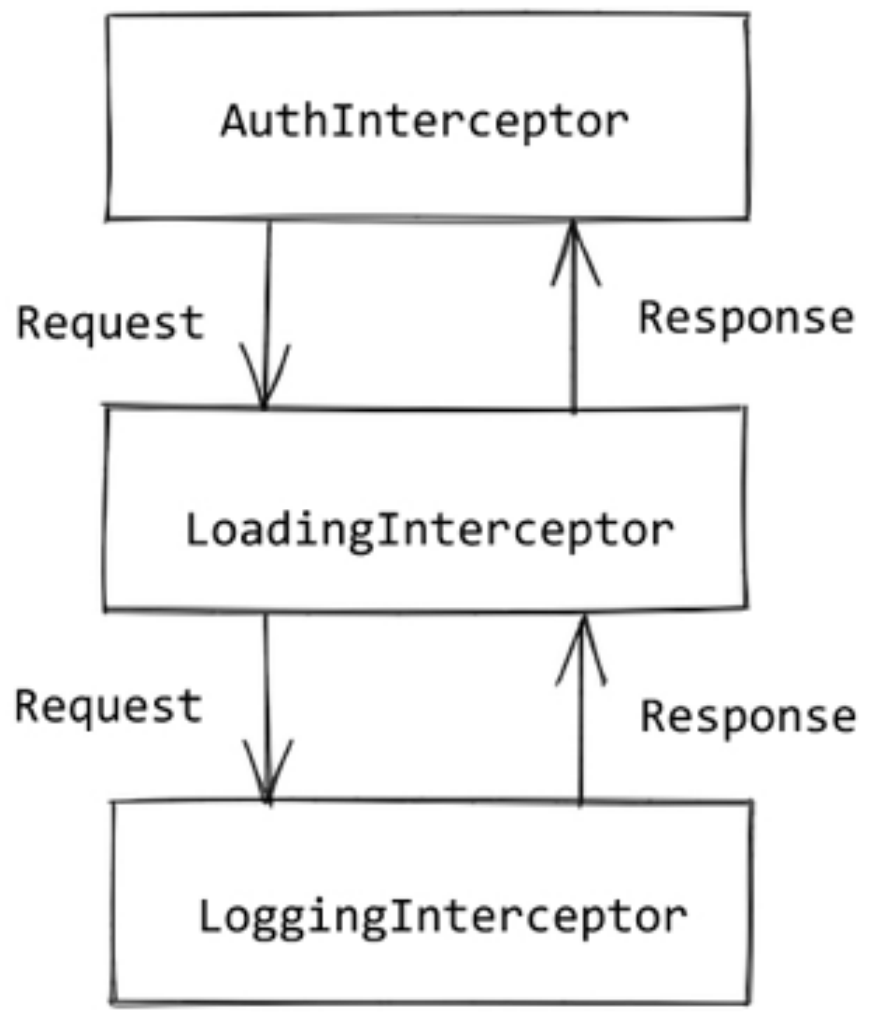

AuthInterceptor
Request
Response
LoadingInterceptor
Request
Response
LoggingInterceptor

Figure 8.9: Execution order of Angular interceptors

By default, the last interceptor before sending the request to the server is a particular service named **HttpBackend**.

Now that we have covered the basics of interceptors, we will use the one we created earlier to set the authentication header for the backend API. Modify the `intercept` method of the `AuthInterceptor` class as follows:

```
intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
  const authReq = request.clone({
    setHeaders: { Authorization: 'myAuthToken' }
  });
  return next.handle(authReq);
}
```

In the preceding method, we use the `clone` method to modify the existing request because HTTP requests are immutable by default. Similarly, due to the immutable nature of HTTP headers, we use the `setHeaders` method to update them.

Angular interceptors have many uses, and authorization is one of the most basic. Passing authentication tokens during HTTP requests is a common scenario in enterprise web applications. However, the token may expire and become useless according to its configuration from the backend server.

In this case, the `auth` interceptor should take this into account and communicate with the `HttpClient` to initiate a request and get a new token. While it is tempting to inject `HttpClient` into the interceptor, it generally should be avoided unless you know what you are doing. You should be very careful because you may end up with cyclic dependencies.

Summary

Enterprise web applications must exchange information with a backend API almost daily. The Angular framework enables applications to communicate with an API over HTTP using the built-in HTTP client. In this chapter, we explored the essential parts of the Angular HTTP client.

We learned how to move away from the traditional `fetch` API and use observables to communicate over HTTP. We explored the basic parts of a CRUD application using the Fake Store API as our backend. We investigated how to implement authentication and authorization in Angular applications. Finally, we learned what Angular interceptors are and how we can use them to authorize HTTP calls.

Now that we know how to consume data from a backend API in our components, we can further improve the user experience of our application. In the next chapter, we will learn how to load our components through navigation using the Angular router.