Angular HTTP

Connecting to the REST API

Abdelfattah Ragab

Angular HTTP

Connecting to backend Rest APIs

Abdelfattah Ragab

Introduction

In this book, I explain everything you need to know about connecting to backend Rest APIs from your Angular application.
In this book, I will show you how to invoke different methods like GET, POST, and the like, how to use interceptors to inject an authentication token into every outgoing request, and much more.
We will cover all areas of calling Rest APIs with Angular.
By the end of this book, you will be able to call Rest APIs from your Angular application in any scenario.
Let us get started.

Understanding communication via HTTP

Most front-end applications need to communicate with a server via the HTTP protocol to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the `HttpClient` service class in `@angular/common/http`.

Providing `HttpClient`

Before you can use the HttpClient in your application, you must configure it.
The `HttpClient` is provided with the help function `provideHttpClient`, which most applications include in the application providers in **app.config.ts**.

```
export const appConfig:
ApplicationConfig = {
  providers: [
    provideHttpClient(),
  ]
};
```

`provideHttpClient` accepts a list of optional feature configurations, to enable or configure the behavior of different aspects of the client.

By default, the `HttpClient` uses the `XMLHttpRequest` API to make requests. The `withFetch` function switches the client to use the Fetch API instead.

fetch is a more modern API and is available in some environments where `XMLHttpRequest` is not supported. It has some limitations, e.g. no upload progress events are generated.

```
export const appConfig:
ApplicationConfig = {
  providers: [
    provideHttpClient(
      withFetch(),
    ),
  ]
};
```

`HttpClient` service

You can then add the `HttpClient` service as a dependency of your components, services or other classes.
It is strongly recommended to create a separate service for API calls instead of calling the backend API directly from a component.
Here are some key reasons for this approach:
Separation of Concerns: Creating a service allows you to separate the logic for retrieving data from the presentation logic in your components. This makes your

components cleaner and easier to maintain, as they focus solely on user interface presentation and user interaction, while the service handles data retrieval and processing.

Reusability: By encapsulating API calls in a service, you can easily reuse the same service in multiple components. This reduces code duplication and promotes the DRY (Don't Repeat Yourself) principle, making your application more modular.

Centralized Error Handling: A service can centralize error handling for API calls. Instead of handling errors in each component, you can implement a consistent error handling strategy within the service. This can include logging errors, displaying notifications or retrying requests.

Improved Testability: Services can easily be mocked or stubbed in unit tests, allowing you to test components in isolation without relying on actual API calls. This leads to faster and more reliable tests.

In the following example, a list of users is retrieved from a backend API.

First, we create a service called `UserService` that will handle all API interactions related to users.

```
// user.service.ts
import { Injectable } from
'@angular/core';
import { HttpClient } from
'@angular/common/http';
import { Observable } from 'rxjs';
import { User } from './user.model'; //
Assume we have a User model defined

@Injectable({
  providedIn: 'root'
})
export class UserService {
  private apiUrl =
'https://api.example.com/users'; //
Replace with your API URL

  constructor(private http: HttpClient)
{}

  // Method to fetch users
  getUsers(): Observable<User[]> {
    return
this.http.get<User[]>(this.apiUrl);
  }
}
```

Note the generic type argument, which specifies that the data returned by the server is of type `User[]`. This argument is optional, and if you omit it, the returned data will be of type `any`.

Normally we use the environment variables to store the server URL similar to this.

```
private apiUrl = environment.baseUrl +
'users';
```

Next, we will use this service in a component to display the list of users.

```
// user-list.component.ts
import { Component, OnInit } from
'@angular/core';

import { User } from './user.model';
import { UserService } from
'./user.service';

// Assume we have a User model defined

@Component({
  selector: 'app-user-list',
  template: `
    <h1>User List</h1>
    <ul>
      @for (user of users; track
user.id) {
        <li>{{ user.name }}</li>
      }
    </ul>
  `
})
export class UserListComponent
implements OnInit {
  users: User[] = [];

  constructor(private userService:
UserService) {}
```

```
  ngOnInit(): void {

this.userService.getUsers().subscribe(
      (data) => {
        this.users = data;
      },
      (error) => {
        console.error('Error fetching
users:', error);
      }
    );
  }
}
```

Making HTTP requests

`HttpClient` has methods that correspond to the various HTTP verbs used to make requests, both to load data and to apply mutations to the server. Each method returns an RxJS observable which, if subscribed to, sends the request and then prints the results when the server responds.
On the following pages I will give you examples of different scenarios of http requests. In some examples I will show you the code of the backend written on NestJS to give you a full idea of how things work, for example when uploading a file or multiple files.
Suppose you are working on a hotel booking application for a brand that has a number of hotels to which the administrator can add new hotels, rooms, facilities, etc.

Users will be able to explore hotels and rooms and make reservations.

GET

`HttpClient` has a get method to read data from the backend.
According to Angular's best practices, you should create a service to connect to the backend.
So here you should have created the HotelsService in the **hotels.service.ts** file, which contains the getAll method as follows

```
getAll() {
  return
this.http.get(environment.API_URL +
'/hotels/all');
}
```

The get method accepts two parameters, the first of which is mandatory, namely the URL, while the second is optional, namely options.
The options parameter is an object with many option configurations. The most commonly used options are headers, params and responseType.

For example, if your backend server is configured to only accept the json format, you need to add the options headers as follows:

```
getAll() {
```