# **Payment Details Application**

**Contents**

[**Payment Details Application** 1](#_Toc153197988)

[**Outline** 2](#_Toc153197989)

[**App Tutorial** 3](#_Toc153197990)

[**Data Privacy ( GDPR )** 4](#_Toc153197991)

[**System Architecture** 5](#_Toc153197992)

[**About the Code –** 7](#_Toc153197993)

[**Future Scope** 16](#_Toc153197996)

# **Outline**

This payment application takes the user’s details and saves them for future payments.

It allows the user to update Create, Update, or Delete an entry from the Database.

This Application can be used in any Application where payment services are required.

**Functionality –**

The Payments App will provide the following functionalities:

1. Create new user accounts.
2. View user details.
3. Delete user accounts.
4. Add new payment details
5. Update existing user information.
6. View existing payment details.
7. Update payment details.
8. Delete payments.

This document describes the technical specifications for the App designed to store user details and perform CRUD operations (Create, Read, Update, Delete) on those details.

# **App Tutorial**

1. To Add Payments Details –
2. Add all the details to the form.
3. Agree to the data consent check switch
4. Sumbit the form
5. Voila !!! Now you can see your payment details on the right side table along with other users.
6. To Edit/Update Details –  
   1. If you need to Update any of the details then just hover on to the update icon next to the name and you’d be able to see your details again in the form.

2. Update the details and submit the form again.

3. You can notice the changes in the table on the right.

1. Delete Details –  
   1. In order to delete a certain entry click on the trash icon in the table

2. Agree to the consent prompt.

3. You can notice the changes in the table on the right.

# **Data Privacy ( GDPR )**

This app is compliant to the GDPR Rules in several ways which are listed below –

1. **Data Minimisation** -  
   Data minimization involves collecting and processing only the personal data that is strictly necessary for the intended purpose. In the context of a .NET Core + Angular application, a data minimization measure has been implemented on both the server side (using .NET Core) and the client side (using Angular).   
     
   **Server-Side (.NET Core):**   
   The data Models in the API only contain important fields and no unnecessary data is being collected.  
   Hence only the necessary data is being transferred between the front end and the back end.  
     
   **Client-Side (Angular):**Form Design – Only necessary input fields are taken for user interaction, thereby removing the use of unnecessary fields.  
   HTTP Requests -When sending requests to the server, only include the data that is required for the specific operation.
2. **Reviewed and Identified Unnecessary Data(HttpResponse)** -  
   While creating an application one should always be very particular about what data they are showing on the screen and what not and this has been taken care of in this Payment Application.  
   Only necessary data like Card Owner’s Name and Card Number is being displayed on the screen and sensitive data like security code(CVV) , Expiration data and even the Balance has been hidden from the users to maintain privacy.
3. **Data Encryption -**Used HTTPS to encrypt data in transit.
4. **Consent Mechanism -**   
   Included checkboxes in the registration form for users to give consent for data processing.

# **System Architecture**

In this Application we have used N-Tier Architecture. This architecture is called n-tier because it has got the Presentation Layer , the Business Access Layer and the Data Layer separated.  
What are these Layers though?   
**Presentation Layer -**The Presentation layer is the top-most layer of the 3-tier architecture, and its major role is to display the results to the user, or to put it another way, to present the data that we acquire from the business access layer and offer the results to the front-end user**.**

**Business Access Layer –**

The logic layer interacts with the data access layer and the presentation layer to process the activities that lead to logical decisions and assessments. This layer's primary job is to process data between other layers.

**Data Layer –**

The main function of this layer is to access and store the data from the database and the process of the data to business access layer data goes to the presentation layer against user request.

**Repository Pattern -**

Repository Pattern has been used in this and this is because the Repository Pattern aims at making your code less tightly coupled.  
Imagine tightly coupled code as if you were to write all of your Web API code in one file only and you don’t do anything to pull it apart so that you can work upon different part of it easily. That is something which is bad and kind of creates a mess. Hence to avoid it we use Repository Pattern.

In a more easier language it is the part of your code where you put database calls. The reason we use Repository pattern in database calls is because we make a lot of database calls and we want something to give us the functionality of **plug and pull.**  
Repository pattern makes your database calls more resuable and also makes your code look cleaner and easier to understand.

# **About the Code –**

The Payments App has been developed using the following technology stack:

Frontend: AngularJS

Backend: ASP.NET Core Framework

### Database: MongoDB **Endpoint definition -**

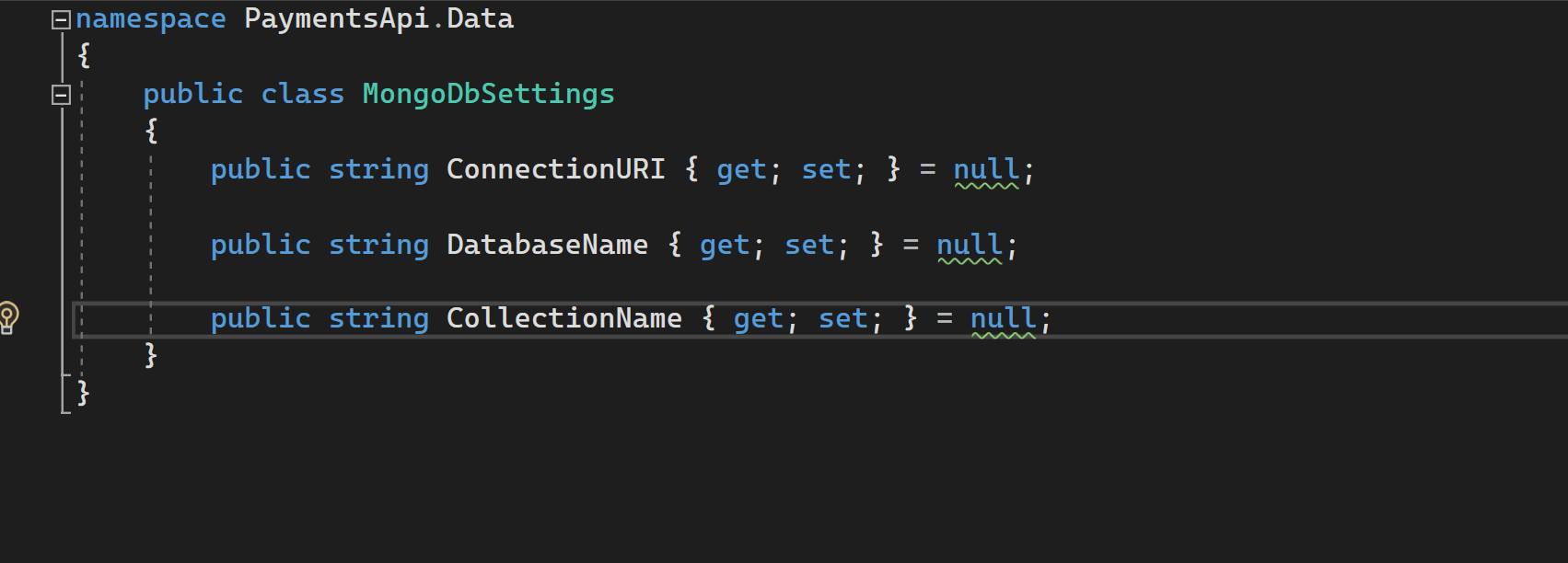
### The Backend API is hosted on URL - <https://localhost:7282>

The Frontend Angular App is hosted on URL - <http://localhost:4200>

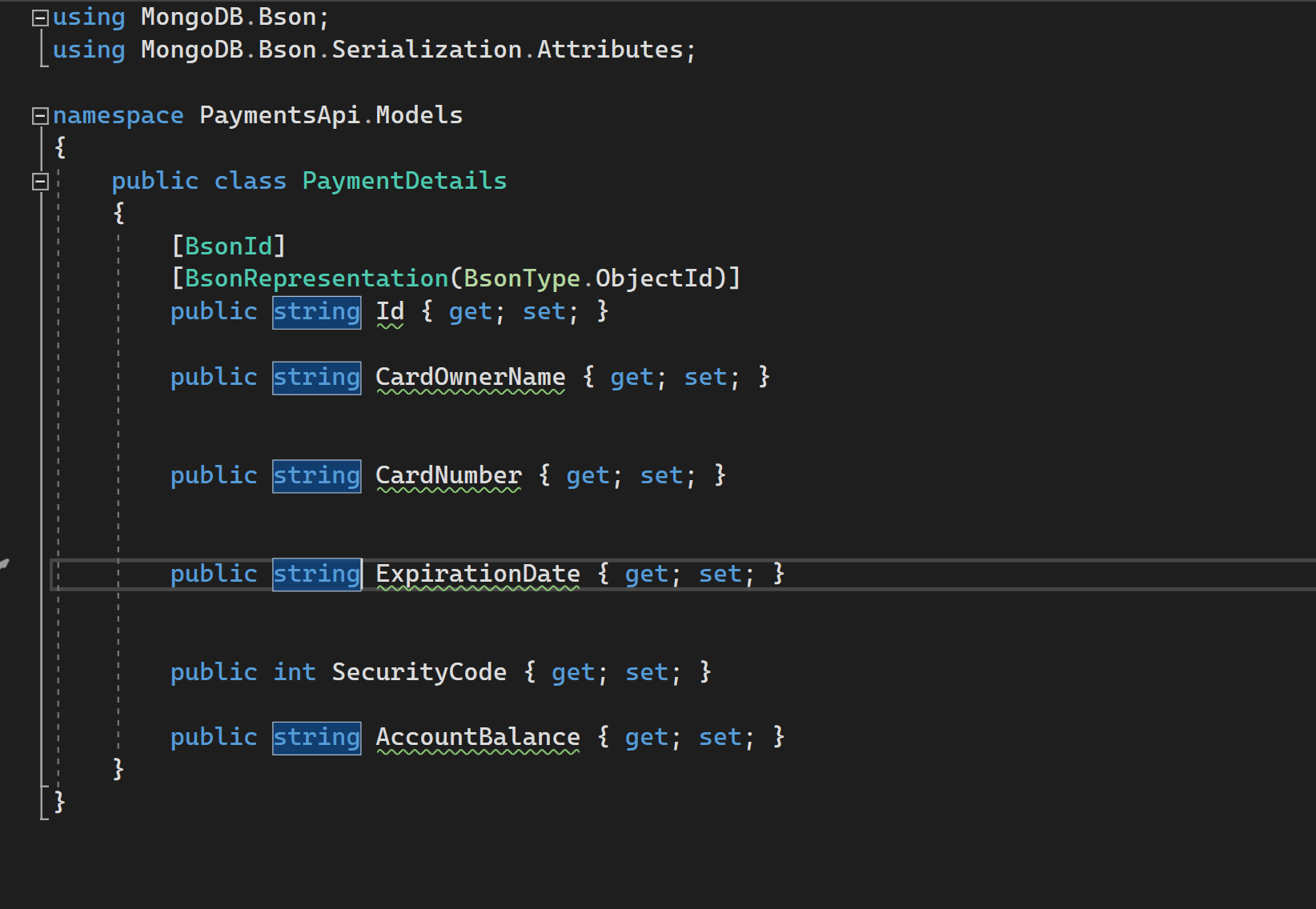
These two are connected using the HttpClientModule in Angular.js and CORS property in C#.

**Backend Code -**The backend API has got 5 major folders namely –

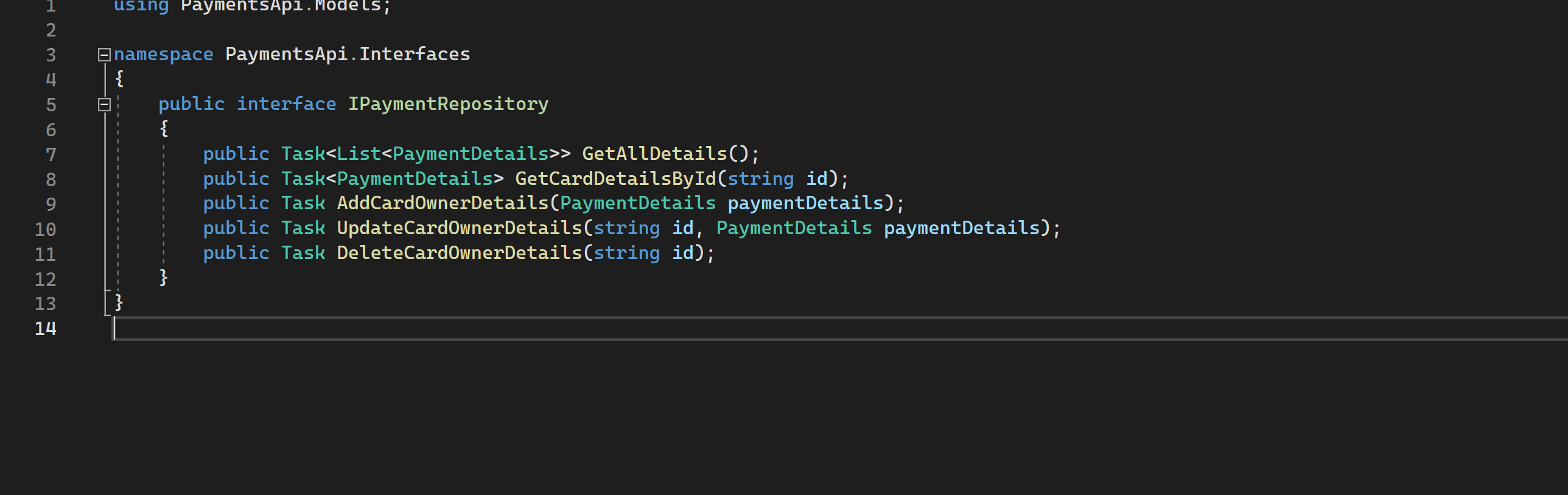
**Data** – The data folder contains class MongoDbSettings which contains the variables we would be configuring the appsettings.json file to establish database connectivity with MongoDB.



**Model** – Model class contains the main fields you want to add to your DB Collection. Consider them as Properties of the Entity for which you are creating the database.



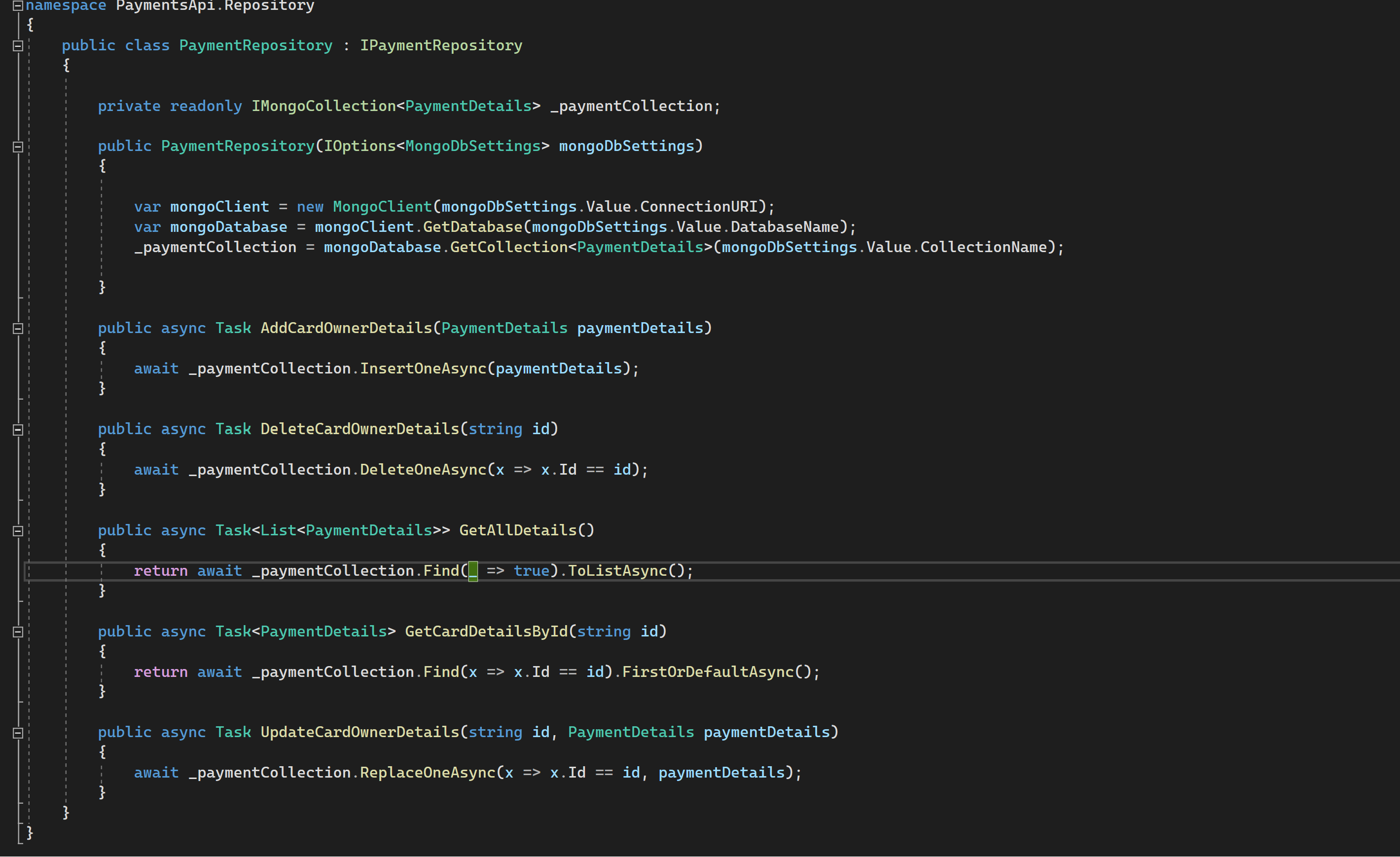
**Interfaces** – Interfaces folder as the name suggests has the interface where methods to fetch data from the database and perform various operations on it.



**Appsettings.json –** We add the configuration details such as Connection string to the DB , Name of the DB , Name of the Collection etc. It’s in this file the program gets to know where to look for the Database.



**Repository** – Repository folder is where the methods of interfaces are implemented.

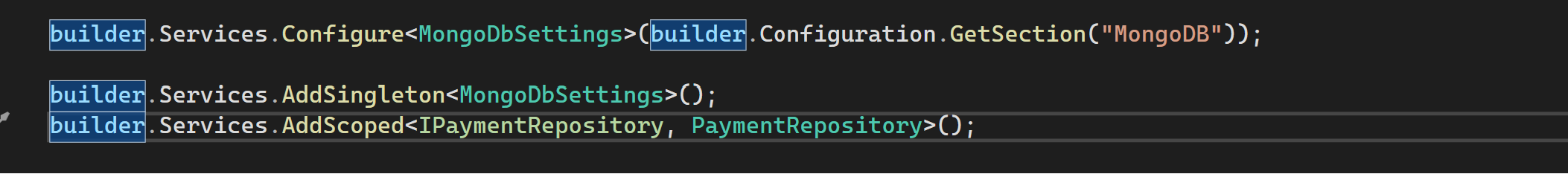


We do not want to call the Mongo API code directly from our controller class and hence we will create a service layer. For this, create a new folder called “Repository”(or you can name it Services it’s up to you) and add a new class called “PaymentRepository.cs” to it as shown above.

Also note that in the constructor we’re taking the values we had passed in **appsettings.json** file and setting them to variables. This would help us to fetch the data from Database.

In the end, the only variable we'll ever interact with for this example is the \_paymentCollection variable.

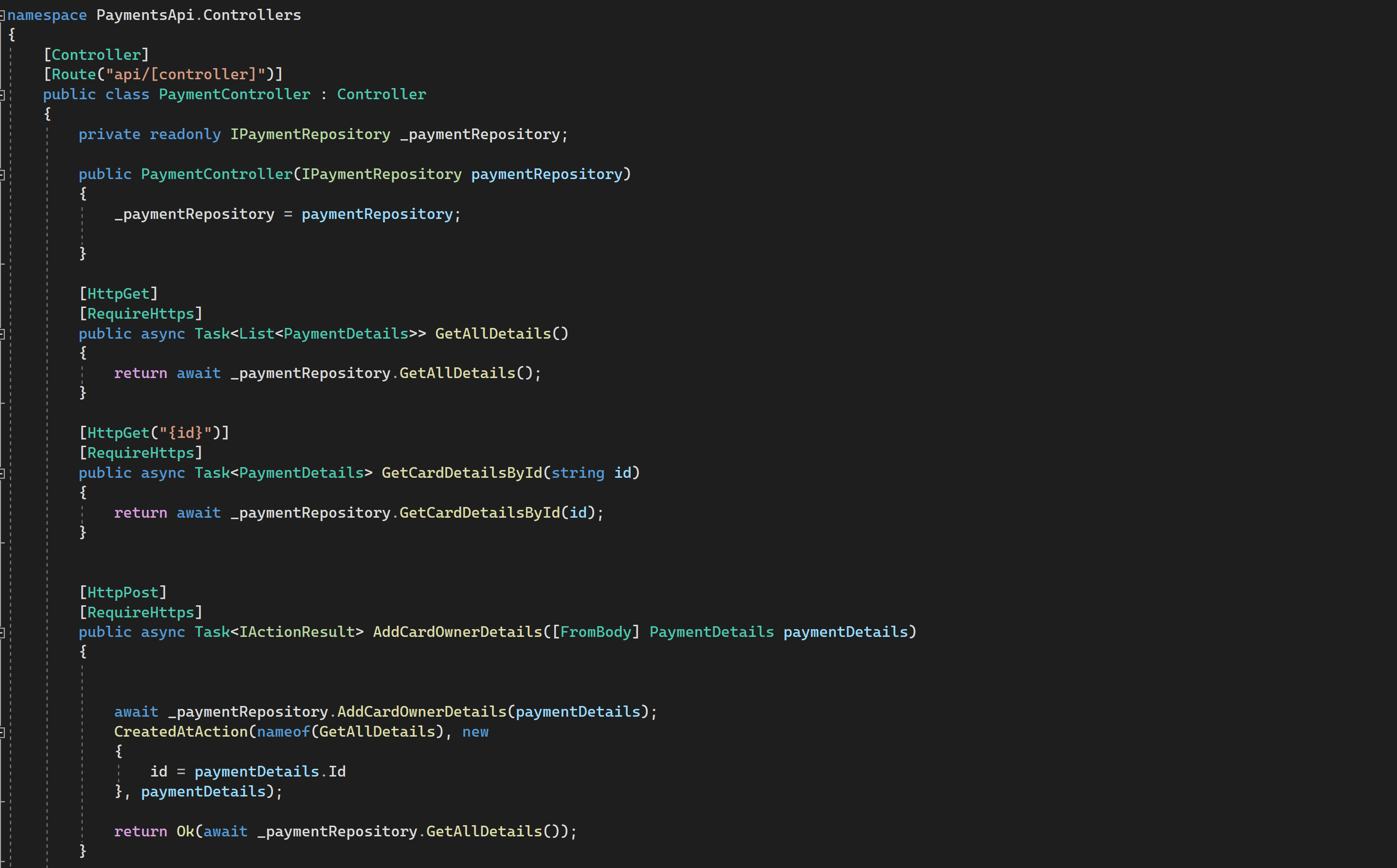
Now we need to connect this Repository/Service to the application and for that we will add the below code to Program.cs file –



**Controllers** – In the Controller we finally call the Repository methods to perform CRUD operations.

In the below PaymentController class, we have a constructor method that gains access to our singleton service class. Then we have a series of endpoints for this particular controller.

Inside the Controller class Based on the incoming request URL and HTTP verb (GET/POST/PUT/PATCH/DELETE), Web API decides which Web API controller and action method to execute e.g. Get() method will handle HTTP GET request, Post() method will handle HTTP POST request, Put() method will handle HTTP PUT request and Delete() method will handle HTTP DELETE request for the above Web API.



This folder structure shows how we can re use the methods anytime anywhere just like the pull and plug analogy.

**Frontend Code –**

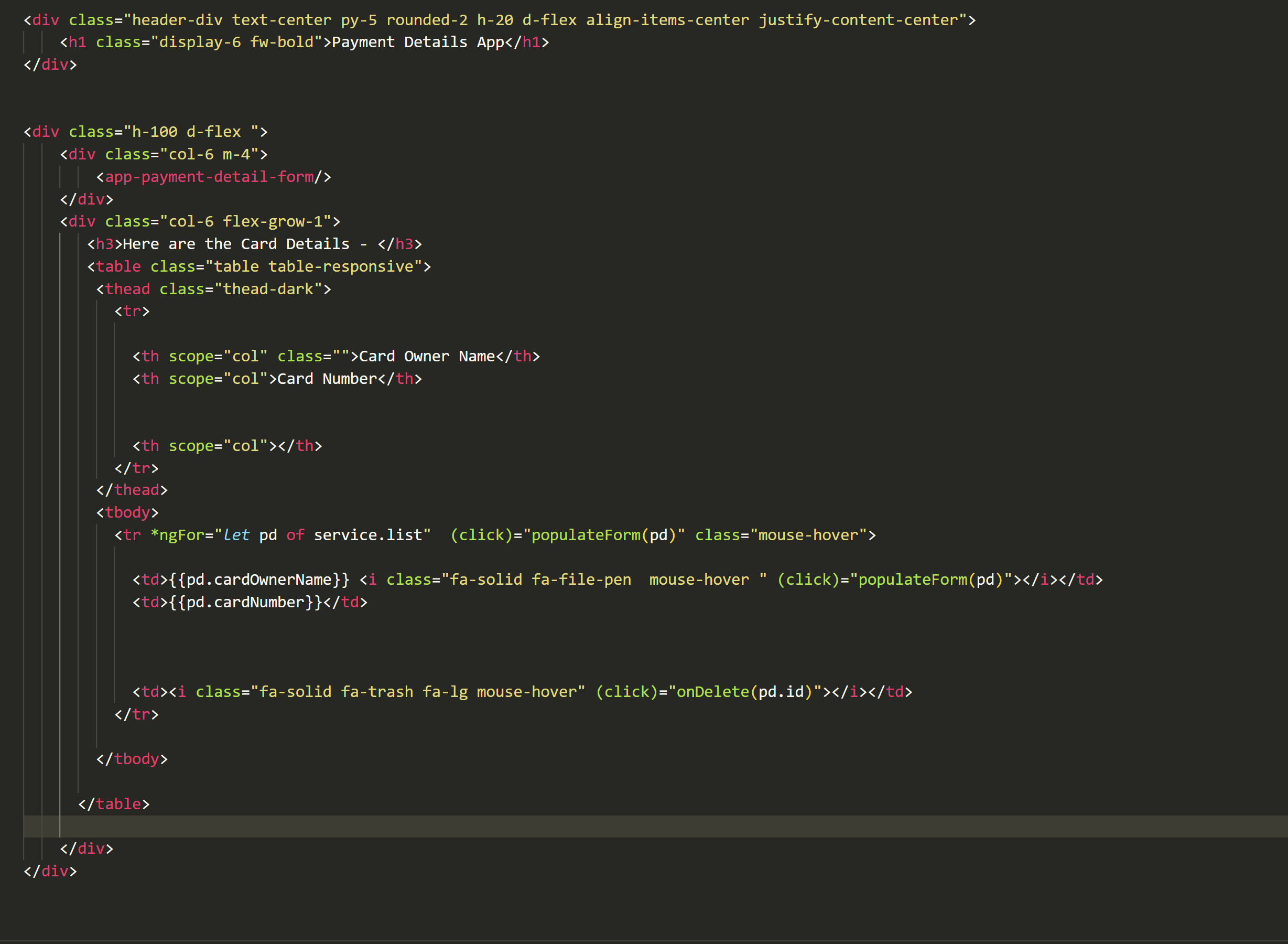
The Frontend of the app has been built using Angular.js.

It has the following Major folders –

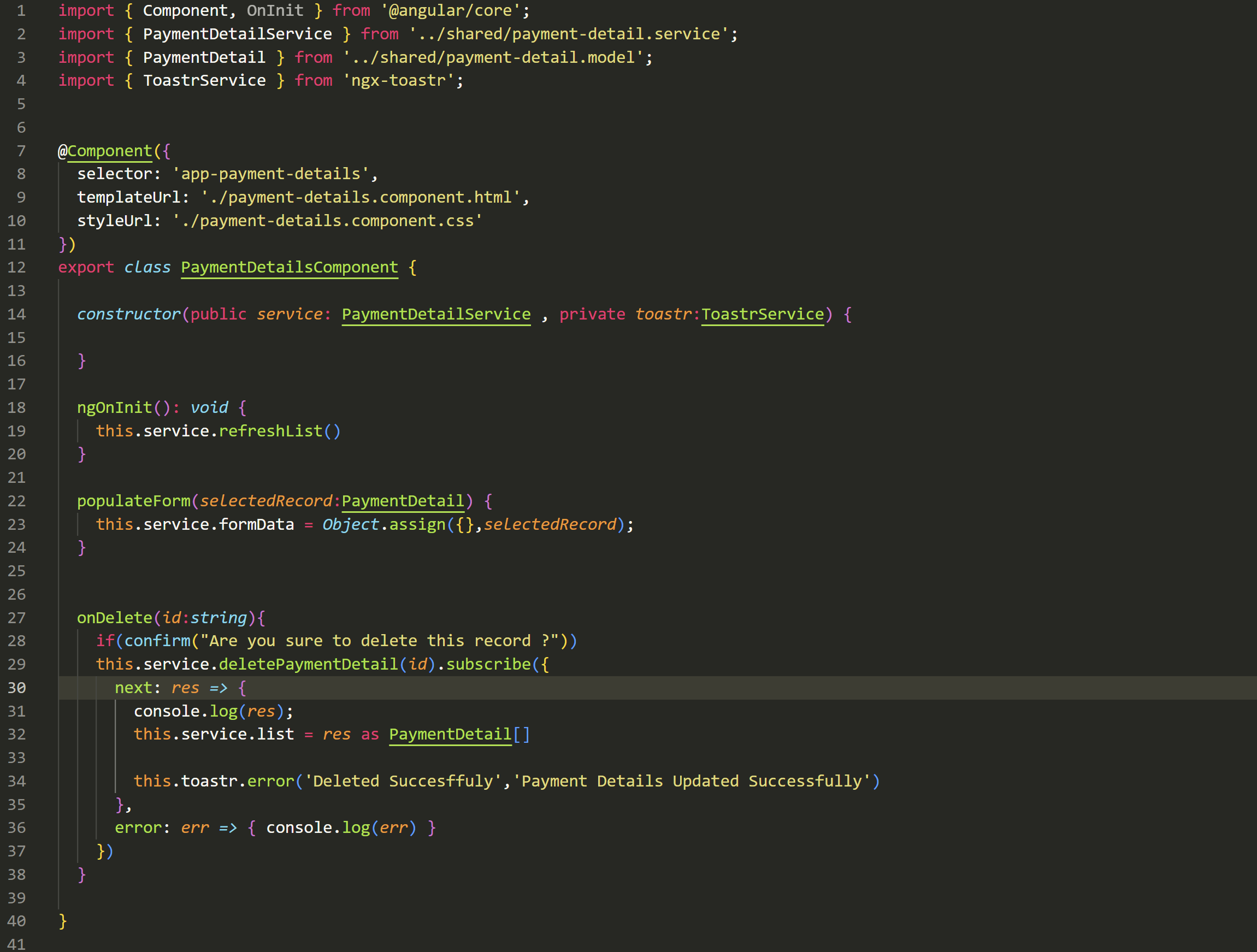
apps – This folder has the PaymentDetail component and a child component called PaymentDetailForm.

Using these two components we have defined the look and feel of the app. Also in the Typescript file we have written the Logic , the methods that define the functionality and user interaction with elements on the screen.

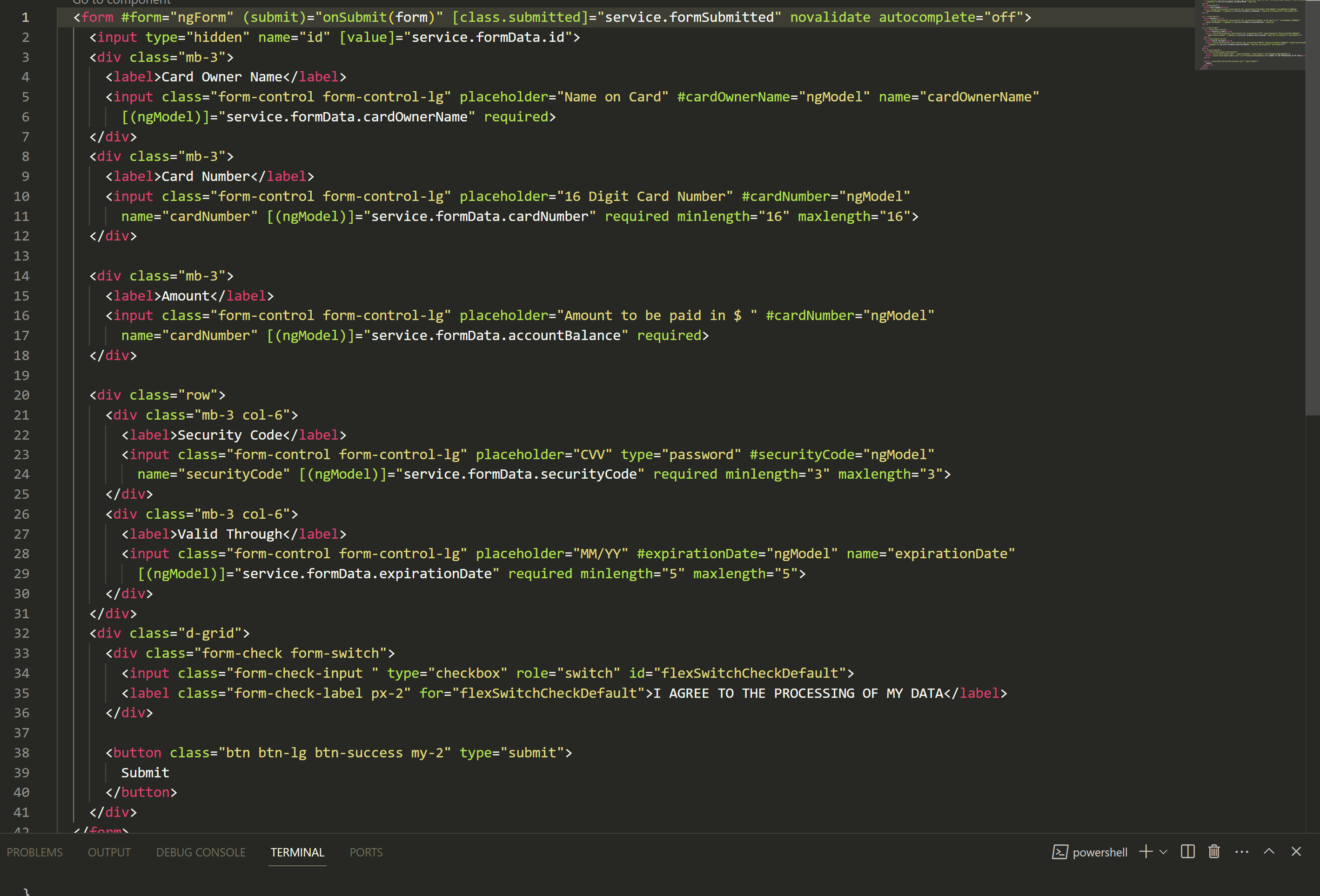
**payment-details.component.html**

****

**payment-details.component.html**



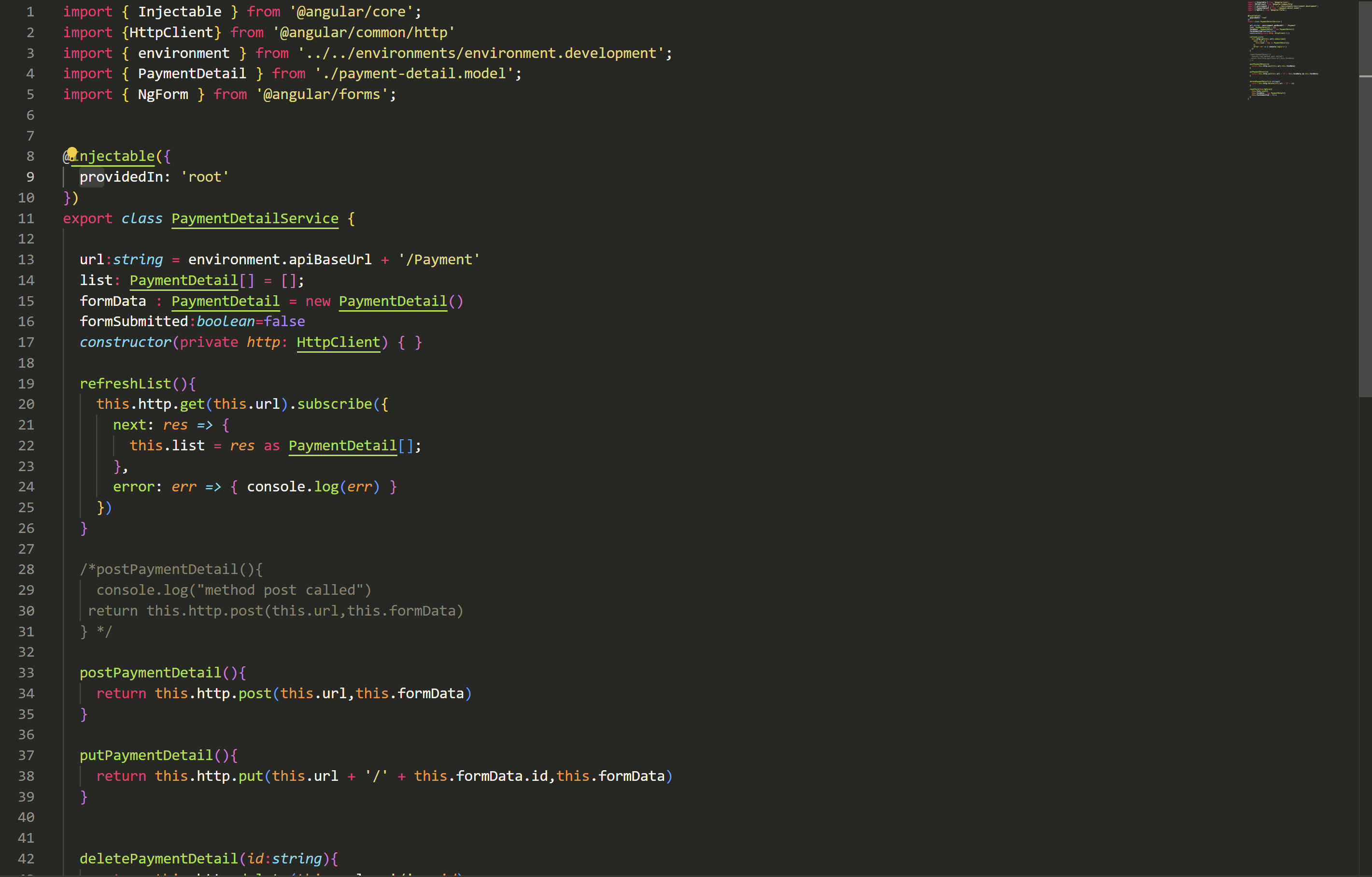
F**orm for input data in the payment-detail-form component**

****

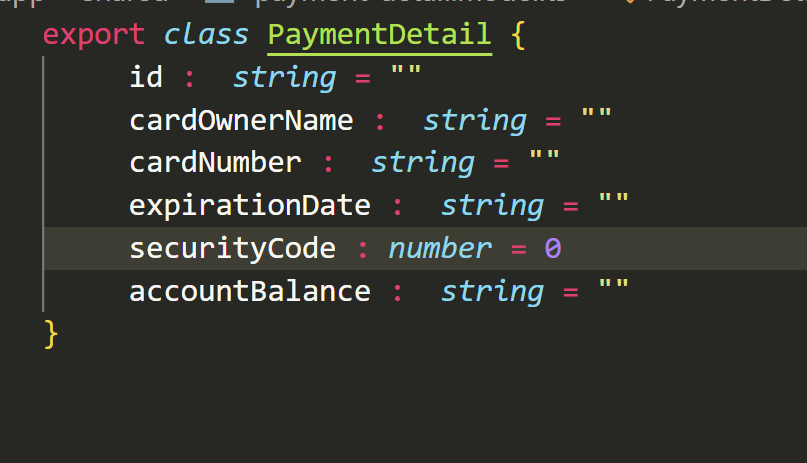
**Services –** The services folder has all the services we implement to add and fetch the data from our API .

Such as refershList is used to bind the GET request , postPaymentDetail is used for POST request and so on.

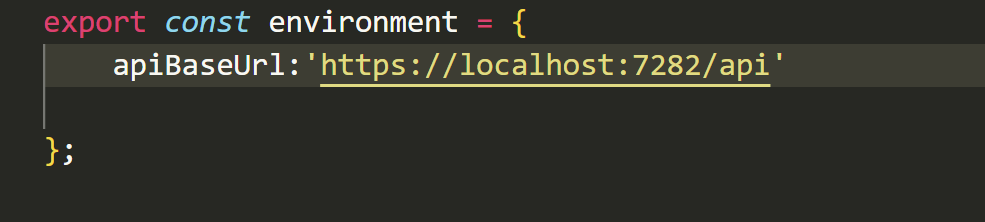
Then from here we can access these methods into our respective components as per our requirements and needs.



**Model –** The model class will help us bind the data to the Model properties we have in our API and hence help us perform operations on it.



**Environment –** In the environment folder we have 2 files one for development with the “development” suffix after it and the other one for production. Inside this class we mention the URL on which our API is running.



**App.module.ts –**

Defines the root module, named AppModule, that tells Angular how to assemble the application.

This is the very first component in an Angular app and where the component structure starts from.

# **Future Scope**

This App is designed to be Modular and Extensible, hence there can be many features added to it, some of them can be –

1. Adding Authentication and Authorization
2. Data Portability-

Allow users to download their data in a machine-readable format.

1. Data Deletion-

Implement a feature for users to request the deletion of their accounts and associated data.

1. Cookie Consent: If your application uses cookies, implement a cookie consent mechanism.
2. Privacy Policy: Clearly communicate your data processing practices and privacy policy.
3. Transaction history.
4. Reporting and analytics.