Payment gateway

https://github.com/ijerosimic/Payment-Gateway

Description

A payment gateway app for submitting payment requests and fetching details of past payments. Includes 2 static html pages serving as a very basic UI.

- The app is a .NET Core 3.1 Web API with C# on the back end and Javascript on the front end
- The app is hosted on Azure and CI/CD is set up via Azure DevOps
- Azure Application insights are enabled and logs are written to Insights using Serilog sinks
- The tests are written using xUnit and Moq
- API endpoints are authenticated via API key in the request header
- An in-memory database serves as a storage mechanism

The Visual Studio solution includes 2 projects.

- Payment Gateway – Web API
- Payment Gateway tests – Test

Folder structure

The web api app consists of 5 folders.

- **Authentication**
  - Contains authentication handler class ApiKeyAuthHandler as well as 2 helper classes for setting up authentication.
- **BussinesLogic**
  - Contains services that perform the core logic of the application.
  - Bank Service class is the endpoint for communication with the acquiring bank. I have decided to mock the bank by using a class named BankEndpoint which produces a fake response when called. This can be replaced with an API call or a message publisher in production.
  - Payment Processor validates the payment data received. There is only a basic validation of credit card number and currency. More validation could be added as necessary.

- **Controllers**
    - Consists of 2 api endpoints.
    - The APIs require a valid API key in the request header.
- **DataLayer**
    - Includes an In-memory database, a seeder method and models.
- **Repository**
    - Contains services that query the database as well as DTOs and extension methods for mapping DTOs to entities and vice versa.

The test app consists of 4 tests classes and a Helpers folder that contains methods for initializing a SQLLite in-memory DB for testing purposes.

Decisions

Single project file

- I have decided to contain the app to a single project because I do not see the benefit of having multiple projects since I will not be reusing these services and I do not have the need to deploy separate assemblies.
- The only reason that I could see for using multiple projects in this solution is the visual and conceptual clarity of the solution. However, I feel that the project folder structure is clear enough without using multiple projects.

Folder structure - Business logic vs Repository

- I have decided to separate services that query the database from services that perform some application logic into 2 folders. I feel it makes the finding the required classes simpler and faster and adds a layer of visual and logical clarity to the application.
- I have decided against having a generic repository is because I feel like queries start to become very unique fast and the generic repository becomes redundant.
- Both folders contain interfaces at the top level to quickly convey the capabilities of each service.

Models

- I have decided to simplify the image of a database and exclude some tables that could otherwise be included. For example, I decided against implementing a separate table for Credit Cards as I felt it would not improve the quality of the solution and instead would add more noise. This would be added in a production database and the credit card data would not be contained in the Payment table.

Tests

- I feel that the tests are the weakest part of the solution partly because of my limited experience with testing as well as limited business logic in the solution. I have tried to test the app as thoroughly as possible while trying to retain the feeling that these tests could work in a production environment.
- Since repository services only query the database the methods are tested against the in-memory SQLite database.
- Controller tests and bank service tests use Moq to provide fake implementations of necessary dependecies. Since both controller and bank service do not containt much logic, they could have been tested only by verifying that the required methods have been called. I am not sure which aproach would be more appropriate so I decided to test that the methods return the valid results.

Assumptions & Improvements

The applications makes some assumptions which would not be suitable for a production app.

- The app assumes that all data received is "clean" and does not do any data validaton. Both front end and more thorough back end validation could be performed especially with deeper domain knowledge.
- Only basic error handling is implemented. As a first step application insights and logging have been enabled, as well as a global exception handler in the Startup.cs class. There are also some basic API response handlers implemented in the front end. A custom exception handling classes might be necessary in a production app.
- The bank endpoint is simulated as a service in the app. This could be replaced by an API call or a message publisher in a production app. I have tried to portray this process to the best of my ability considering limited domain knowledge.
- The repository methods are asynchronous under the assumption that these would be very complex queries in a production app.
- Bank service is also written as an async method since it is presumed that it contacts the bank API which requires a certain amount of time to process the payment.
- I have decided to leave payment processor as synchronous since it does not contain much logic and it computes relatively fast. Depending on the complexity of the logic in a production app the processor could be written as an async method.

I hope you enjoy this app!