Payment Gateway

Location

https://github.com/mjrp11/PaymentGateway

Project Structure

Solutions Location: https://github.com/mjrp11/PaymentGateway/tree/master/src

- **Config folder:** This folder contains the global configuration settings to be used by the application. You can change the configurations inside the *"globalsettings.json"* file (each config explained later on), or you can create a new file called *"globalsettings.<Anything>.json".* You can then create an environment variable named *"ASPNETCORE_ENVIRONMENT"* with value "*<Anything>"* for the code to read this second file as well.
- **PaymentGateway.Api:** This folder contains the project for the configuration of the api. This project will contain the setup of the api (configuration, logging, inversion of control, etc), as well as the controllers for the application. If we needed to change the way the users interact with the application, we would only need to change this layer (or add a new one if we wanted to have more than one way, like a message consumer application).
- **PaymentGateway.Application:** This folder contains the project for the application. This project contains all the logic required to process the requests made to the controller. If we needed to completely change the way the application works, we would only need to change this layer.
- **PaymentGateway.Client:** This folder contains the project for the client. This project contains the client so that its easier to integrate this service in another .NET solutions. (You can see how by checking the End-to-End Tests project).
- **PaymentGateway.Domain:** This folder contains the project for the definition of domain entities. This project is used to define the basic DTO needed by the application.
- **PaymentGateway.E2ETests:** This folder contains the project for the End-to-End Tests project. This project allows us to test a running/deployed application.
- **PaymentGateway.IntegrationTests:** This folder contains the project for the integration Tests. This project allows us to test the solution (from the controller to the DB) without having to interact with any deployed systems. By creating a new in memory DB in each execution and mocking the Bank interaction, we can test every flow of our solution without interacting with external applications.
- **PaymentGateway.Migrations:** This folder contains the project for the database migrations. This project allows us to create all the required database structed needed to run this application. To

run the migrations its possible to simply run this project (as a console application) or run *"dotnet ef database update –context Microsoft.EntityFrameworkCore.DbContext".*

- **PaymentGateway.SqlLiteDbConnection:** This folder contains the project for the sqlite project. This project allows us to use a SQLite DB for this solution.
- **PaymentGateway.SqlServerDbConnection:** This folder contains the project for the sqlserver project. This project allows us to use a SQL Server DB for this solution.
- **PaymentGateway.UnitTests:** This folder contains the project for the unit tests project. This project allows us to test each individual component of the application separately.
- **PaymentGateway.ViewModels:** This folder contains the project for the view models project. This project allows us to share the entities required to interact with the application from external sources.

How to run this project

The first step to run this project is to look into the configuration files. The most important one is the *"globalsettings.json"* file. Like explained previously you can either change this file directly or create a new file in the same folder as *"globalsettings.<Anything>.json"*. I recommend the second option named something like *"globalsettings.checkout.json"*.

As you can see this file is a *json* file containing a set of values. The j*son* structure must remain the same, although the values can (and should) be changed. The values needed are:

- *Database Type:* This setting indicates which database engine we want to use (usually our application would only use one type of database, I only included 2 to show how easy it would be to switch the engine in case of need). The accepted values are *"SqlServer"* or *"SqlLite".*
- *Database SqlServer ConnectionString:* This setting is the connection string in case we selected *"SqlServer"* on the database type. This value should be something like: *"Data Source=127.0.0.1;Initial Catalog=PaymentGateway;User Id=DBUsername;Password=DBPassword"*
- *Database Sqlite ConnectionString:* This setting is the connection string in case we selected *"Sqlite"* on the database type. This value should be something like: *"Data Source=<PathToFolder>\\<FileName>"*
- *Encryption Key:* This setting can be any random string and it will be used by the application to encrypt the sensitive data. Since the code is not currently using any decrypt operation (see Assumptions) you can change this value for each run, however if you needed the values to be decrypted this value could not change between runs.
- *Encryption SSM_Path:* This value is not needed. This value was simply used to indicate what would be required if we used AWS SSM instead of getting the encryption key from the configuration file (see Problems/Limitations)

If you set up your configuration file directly on the *"globalsettings.json"* file you can already start the application. However, if you created a new file you need to set up the environment variable

“ASPNETCORE_ENVIRONMENT” with the value that you put on your configuration file (on the example I gave this would be *“ASPNETCORE_ENVIRONMENT=checkout”).*

If you want you can also setup the API appsettings file the same way that you did for the global settings one (either change the file directly or create one with the same extension). This file allows to setup the console logging of the application and even setting up a file logging as well.

Once you are done setting up the configuration file you can run the “PaymentGateway.Api” project (however I would recommend setting up the environment variable “*ASPNETCORE_URLS”* to something like “*http://localhost:2345”* so that the application always runs on the same port).

After the application is running you can make a rest api post call to *“http://localhost:<port>/PaymentGateway”* in order to make the process payment request, or you can make a get call to *“http://localhost:<port>/PaymentGateway/<PaymentID>”* to make the get payment request.

For the process payment request you need to have a message body like: *{"CardNumber":"4111111111111111","ExpireDateYear":2021,"ExpireDateMonth":10,"CVV":"123","Amount":10.23,"Currency":"EUR"}.* If all the values are valid, this method will return a response similar to: *{"status":0,"isSuccess":true,"paymentId":"72cdea49-2251-4f50-9090-b3e3d8251981"}.*

Simulating the bank

For simulating the bank I created the class FakeBankGateway. This class will return a success payment 90% of the time, a *“LackOfFunds”* error 5% of the time, an *“InvalidCard”* error 4% of the time and an *“UnexpectedError”* error 1% of the time.

In order to create a proper bank connection it would be required to create a class that would need to extend the IBankGateway Interface and change the inversion of control to use the new class.

Assumptions

- This project assumes that the there is no major limitation in the time required to process a payment, and that the card mask is not information sensitive enough that should not be on the DB. This decision was made to save time on the *Get Payment* request. This way there is no need to decrypt the card the number and construct the card number mask each time a *Get Payment* request is made, saving time for the clients if the number of get operations is superior to the number of process operations. If we needed to ensure that the process request is the most performance as possible (at the expense of the get request), or if it is not possible to save card number mask on the database for legal reasons then this logic would need to change.
- It was also assumed that the card expiration date is not sensitive information.

Problems/Limitations

- This implementation assumes a card is unique if the *“card number”, “CVV”, “ExpireDateMonth” and “ExpireDateYear”* are the same. Which means that for this application the following cards:
    - 4111111111111111 10/2021 123
    - 4111111111111111 10/21 123
    - 4111111111111111 10/21 321
    - 4111111111111111 11/21 123

  Are all different. There should be way to update card information to ensure that cards with the same *“card”* number are the same.
- The current solution gets the keys to encrypt and decrypt the sensitive information from the configuration file. This solution is not the most secure. The is on the code a class named *“SSMEncryptionKey”* that intends to use AWS SSM as the solution to get this key, however in order to limit time/cost I did not create an AWS account in order to test this class.
- The number of unit, integration and End-to-End Tests are limited. They were created as an intention to show how each test differs from one another and not to create a full test coverage of the solution.

Extra Mile

- **Application logging:**
    - The application allows logging on the console and onto a file (if activated on the appsettings file of the api). More loggers configuration can be added to the *ConfigureLogging* method the api program.
- **Containerization:**
    - The file RunningOnDocker.ps1 can be used to run the application on docker. Its recommend to either change the globalsettings configuration file directly or create a new one called “globalsettings.docker.json”. This file will publish the API project, build the docker image, and run the image exposing the port 2345. (You can use *host.docker.internal* if you need to reference your localhost inside the application, for instance for the connection string of the sql server).
- **API client:**
    - The file PaymentGateway.json was exported from the tool Talend API Tester on chrome. You Should also be able to import it into Postman.
- **Build script / CI:**
    - On the github repository you can check the CI pipeline on the tab “Actions”. The CI pipeline restores the packages, builds the application and runs the unit and integration tests (it does not run the End-to-End tests because they need a deployed application).
    - In order to convert this from a CI to a CD pipeline we could use 2 options:
        - Build the docker image and push it into an image repository, and use that to deploy the image as a containerized solution (on Kubernetes for example).

        - Create a nuget package, extract the package on a server and deploy it (using IIS for example).

- **Encryption:**
    - The encryption was only used to encrypt the card number and card CVV into the DB. However, it could also be used to encrypt the configuration file.
- **Data storage:**
    - This application uses either sqlite or sql server, however any relation database could serve this application.