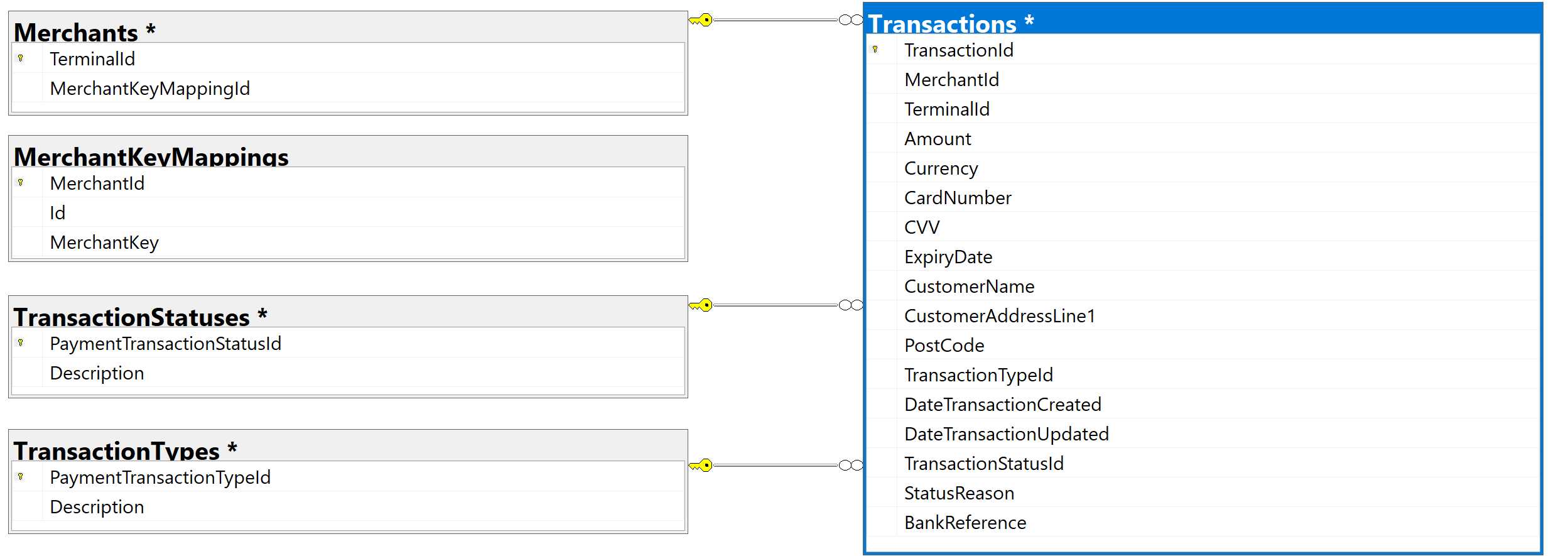
Overview of PaymentGateway

# Section 1: Data Model



The above diagram depicts the tables and their relationships in the database.

### Table: MerchantKeyMappings

Each merchant will have a unique API key. The API key is used to authorize the merchants. The API key is presented to the PaymentGateway API in HTTP request header, called “merchantKey”.

### 1.2 Table: Merchants

Each merchant will have at least one terminal associated with. While processing a transaction, the merchant should specify the name of the terminal the transaction is originated from. All terminals that merchant can post transactions to should be registered in this table. If a merchant is trying to post a transaction for a terminal which is not registered in this table against that merchant, the merchant will get HTTP 403 Forbidden response.

A merchant may have more than one terminal. The Mechants.MerchantKeyMappingId is a foreign key to MerchantKeyMappings.Id

Note: This is something I have come up. This may not be applicable in a real-world ecommerce world.

### 1.3 TransactionStatuses

This table holds a list of valid statuses that a transaction can have. Currently, valid transaction statuses are

|  |  |
| --- | --- |
| PaymentTransactionStatusId | Description |
| 1 | Succeeded |
| 2 | Failed |
| 3 | InProgress |
| 4 | Exception |
| 5 | Cancelled |
| 6 | Error From Acquirer |

The Transactions.TransactionStatusId is foreign key to TransactionStatuses.PaymentTransactionStausId

### 1.4 TransactionTypes

This table holds a list of valid transaction types. Currently, valid transaction types are

|  |  |
| --- | --- |
| PaymentTransactionTypeId | Description |
| 1 | CreditCardPayment |
| 2 | DebitCardPayment |
| 3 | PreAuth |
| 4 | Auth |
| 5 | Refund |
| 6 | Cancel |

The Transactions.TransactionTypeId is foreign key to TransactionStatuses.PaymentTransactionTypeId

When a merchant uses a transaction type that is not found in the able, it results in HTTP 400 Bad Request.

### 1.5 Transactions

This is the master table that holds details of transaction sent to PaymentGateway by merchants. This table stores all transactions irrespective of the transaction result.

All fields except DateTransactionUpdated, TransactionStatusId,StatusReason,Bank Reference are stored in the table before being forwarded to Acquirer. A unique TransactionId, which is SQL Identity column generated to reference the current transaction.

DateTransactionUpdated, TransactionStatusId,StatusReason,Bank Reference are updated upon receiving a confirmation response from Acquirer. The Acquirer may have approved/declined the transaction. In either case, we respond merchant with the final status of the transaction whether it’s succeeded or failed and the reason for failure.

# Section 2: Details Merchants, Terminals and API Keys to use to work with API endpoints

I have already setup few merchants that we can use to Post/Get transactions.

|  |  |  |  |
| --- | --- | --- | --- |
| MerchantId | MerchantKey | TerminalId | TransactionId |
| M0001 | hCQcGPPOf0DKCLSTWDlveQ== | T0001 | 10000 |
| M0001 | hCQcGPPOf0DKCLSTWDlveQ== | T0001 | 10001 |
| M0001 | hCQcGPPOf0DKCLSTWDlveQ== | T0002 | 10002 |
| M0002 | 7UWuP2WPizfr4O7GmepI3A== | TM0001 | 10003 |
| M0003 | eDhl3PO6DZ/lX1Ype9tbFQ== | TMX001 | 10004 |

Merchant “M0001” can use either of the Terminals “T0001” or “T0002” to post a transaction.

The database is also seeded with few transactions against the registered merchants that can be used to test get transaction detail end point.

Where as the other two merchants currently have only one terminal registered for them. In total we have 3 merchants to work with.

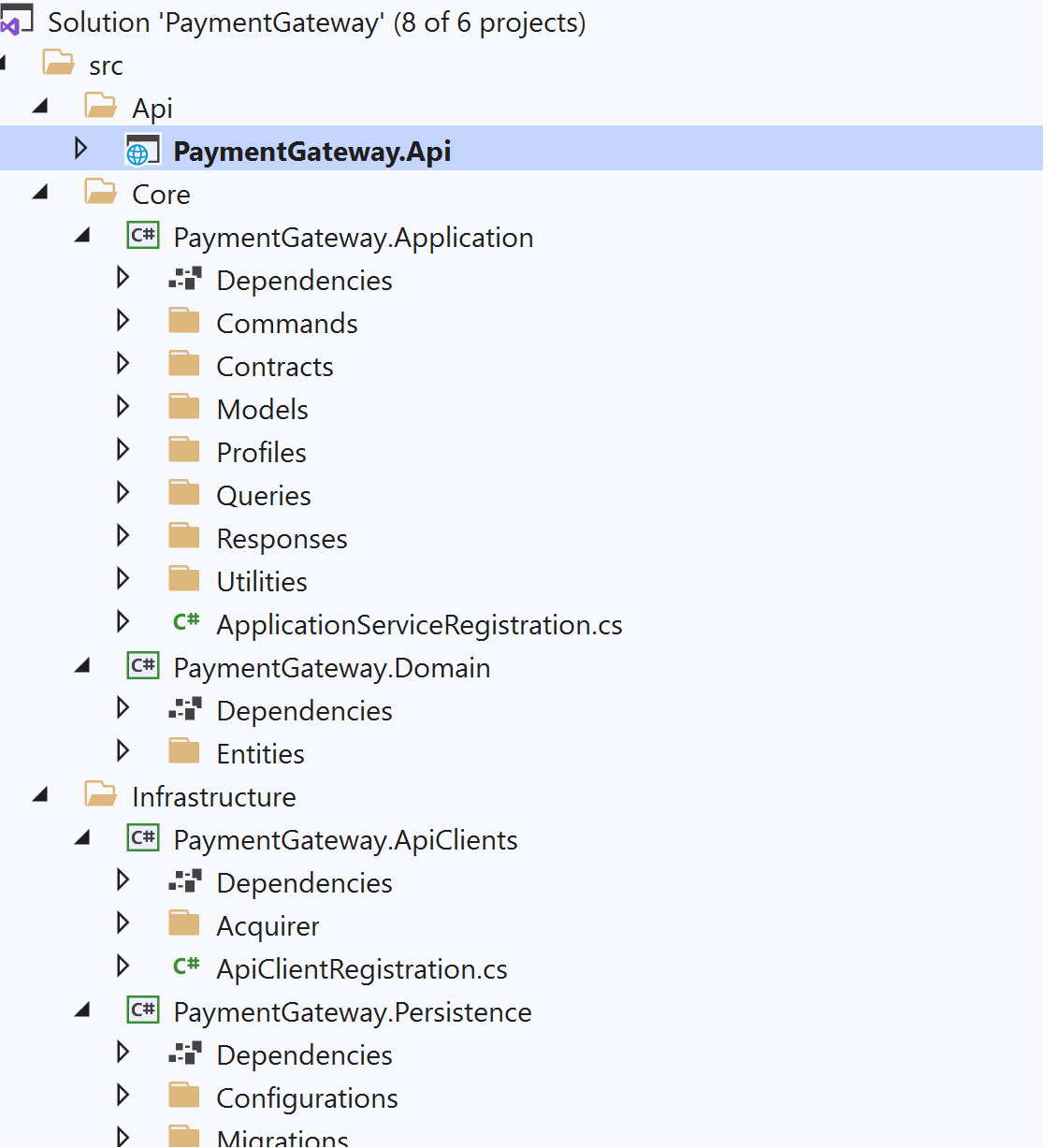
# Section 3: Application Design/Project structure & components

The application design follows Onion Architecture (A.K.A Clean Architecture) principles.

## Section 3.1: .Net core versions

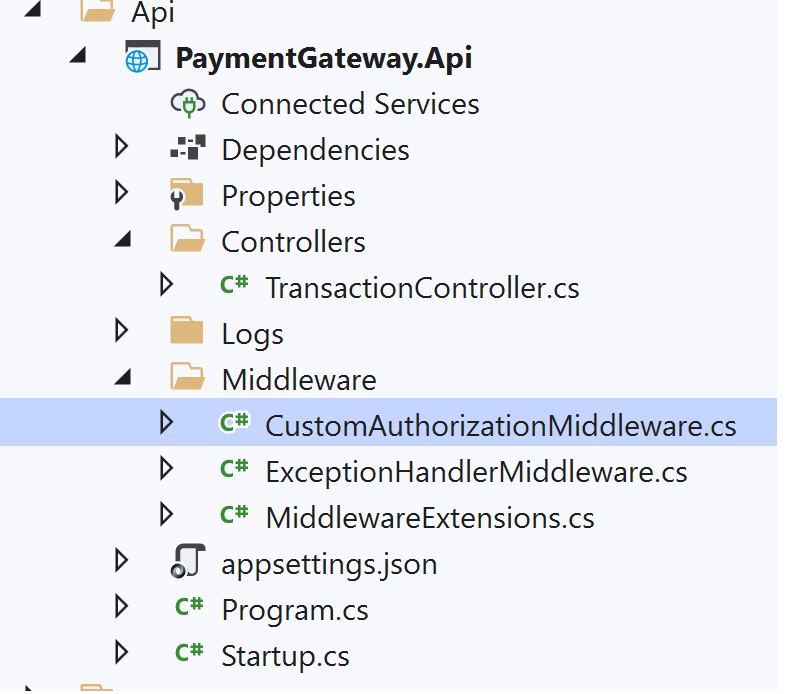
The PaymentGateway API has been developed using .NET Core 3.1. The supported class libraries are developed using .Net Standard 2.0

## Section 3.2: Application Design/Project Structure



### Section 3.2.1: PaymentGateway.Api

Built in .Netcore 3.1, this project follows typical API design. The project hosts the startup class to register various services for dependency injection, setup logging, controllers and custom middleware.



#### 3.2.1.1 Controllers

There is one controller, TransactionController.cs which has a GET endpoint to retrieve existing transaction detail. And a POST endpoint to post a new transaction

Get returns either a OK result (a transaction found) or NotFound result (transaction not found)

Post returns either a Created result (transaction processed successfully) or BadRequest(invalid payload)

The controllers are lightweight. They just pass the control to the hand handlers through Mediator pattern and builds a action result based on response.

#### 3.2.1.1 Middleware

The CustomAuthorizationMiddleware.cs component is responsible to check if there is “merchantKey” present in the header and is a valid one. Otherwise it throws 401 Unauthorized result

The ExceptionHandlerMiddleware is responsible to handle any unhandled exceptions in the pipelines.

All of these middleware components have been added to HTTP pipeline

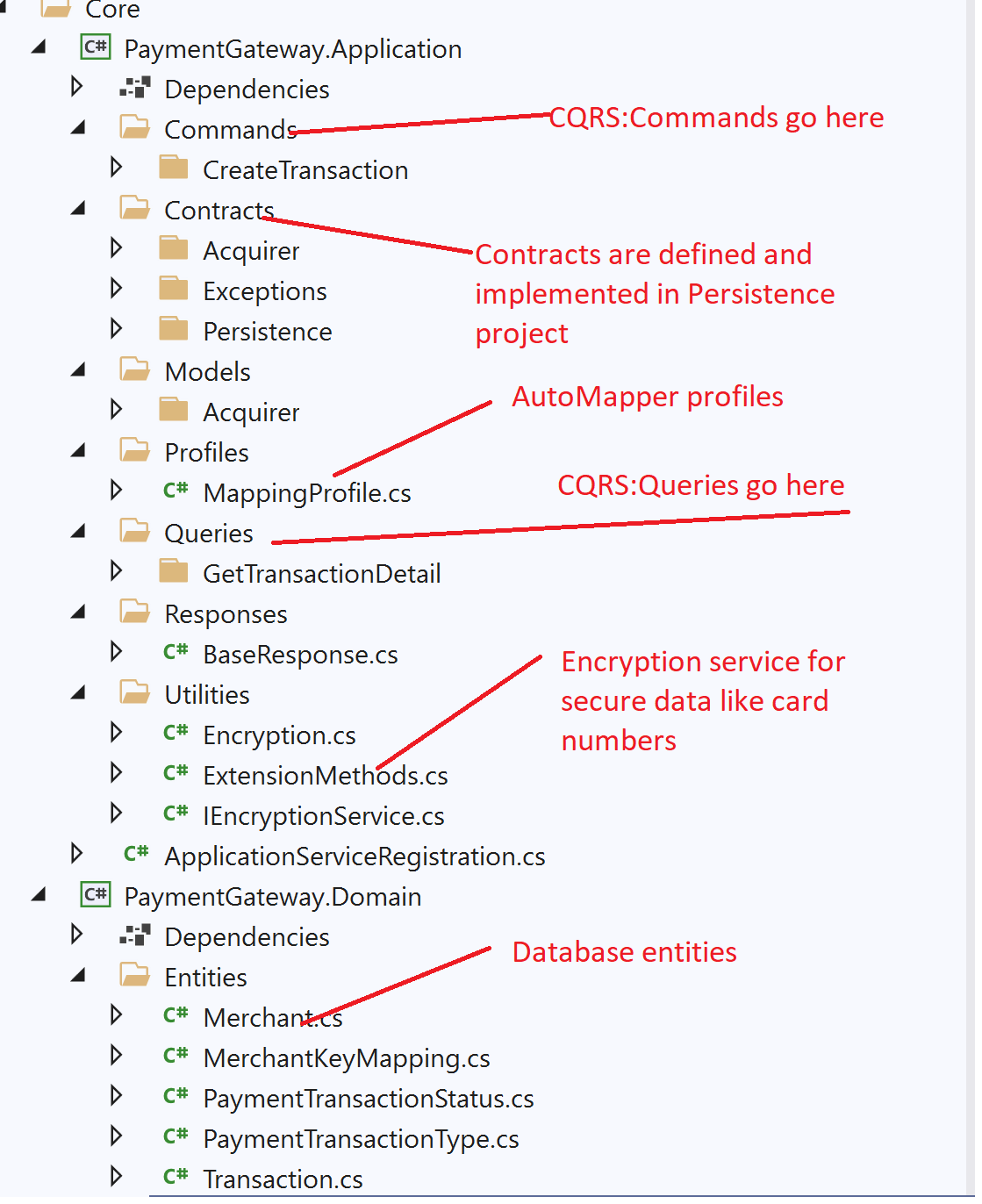
The appsettings.json has settings for

* . ConnectionString
* .Log customisation
* Acquirer API settings
* Key for encryption/decryption of CreditCard numbers and CVV

#### 3.2.1.3 Logging

The application uses Serilog and has been configured to output the messages to the console. It can be customized through appsettings

### Section 3.2.2: PaymentGateway.Application



The PaymentGateway.Application is core of the application. Much of the business logic goes here and has been built on .Net Standard 2.1.

The application follows CQRS pattern. The command to create transaction is implemented in a different class and the query to get transaction details in another class

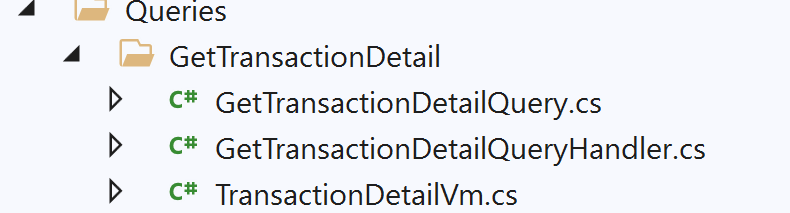
#### Section 3.2.2.1 Commands



Using the MediatR pattern, the “CreateTransactionCommandHandler.cs” is the implementation of IRequestHandler. This class does following jobs

1. Using the FluentAssertions library defined in “CreateTransactionCommandValidatior.cs” , validates the input request is valid.
2. Checks if the merchant is authorised to post the transaction against the terminal specified in the request.
3. Stores the transaction in database, encrypting the secure data like card numbers and CVV and forwards the request to Acquirer. Upon receiving the response from Acquirer , updates the transaction with the result and sends the response back to the caller (controller in this case)

#### Section 3.2.2.2 Queries



Using the MediatR pattern, the “GetTransactionDetailQueryHandler.cs” is the implementation of IRequestHandler. This class does following jobs

1. Checks if the transactionId belongs to merchant that has been authorized.
2. If yes, gets the transaction detail from database, decrypts and masks the sensitive data and returns the response to caller (controller in this case)
3. If either transactionId is not found or doesn’t belong to the merchant, sends NotFound response.

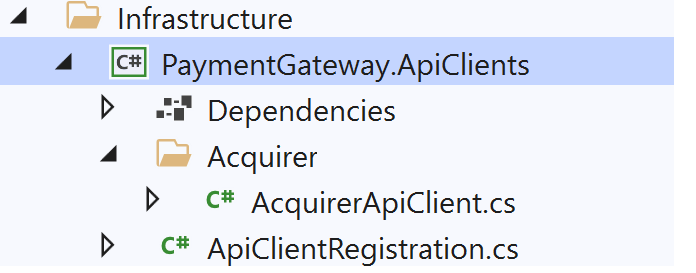
### Section 3.2.3: PaymentGateway.Domain

Being part of “Core” application is the Domain project. This project hosts the entity object model used for EF Core data model.

1. Merchant => dbo.Merchants table
2. MerchantKeyMapping => dbo.MerchantKeyMappings table
3. PaymentTransactionStatus =>dbo. PaymentTransactionStatuses table
4. PaymentTransactionType => dbo. PaymentTransactionTypes table
5. Transaction=>dbo. Transactions table

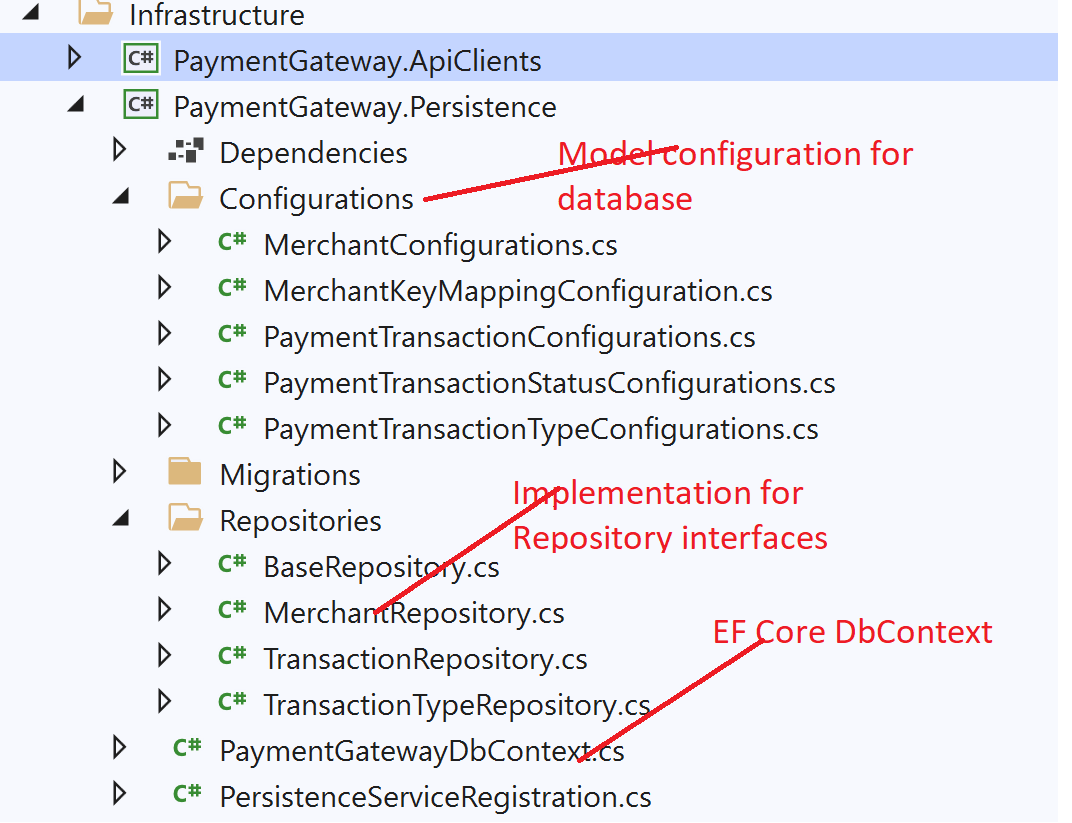
### Section 3.2.4: PaymentGateway.ApiClients.

This project being part of Infrastructure, has the implementation to talk to the acquirer/bank. Currently it talks to mock Acquirer. It can be easily replaced to talk to a real acquirer with/without changes



### Section 3.2.5: PaymentGateway.Persistence.

This project being part of Infrastructure, has the implementation to talk to Database. This project uses EF Core Code-First model and has the implementation for the repository contracts defined in the PaymentGateway.Application project



The EF Core DbContext class “PaymentGatewayDbContext.cs” class has the code that creates the database if one doesn’t already exist and seeds the database. Please refer to Section 1: Data Model to see about the details of seeded data.

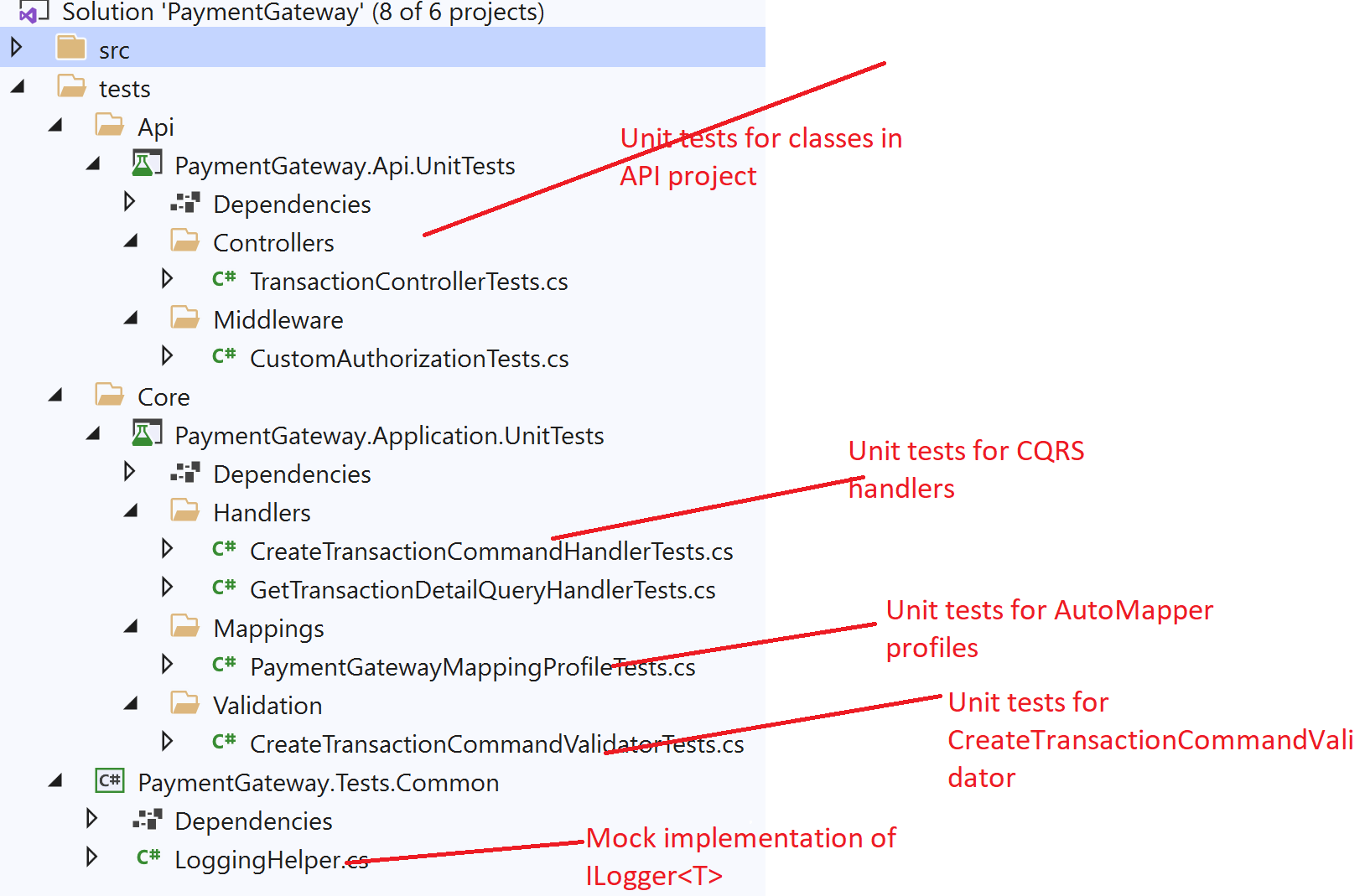
## Section 3.3: Database

In Startup.cs, the code below ensures the a database is created with name PaymentGatewayData in (localdb)/MSSQLDB and seeds with initial data, if doesn’t already exist.Based on the settings in appsettings.json

## 

## 

## Section 3.4: Unit Tests

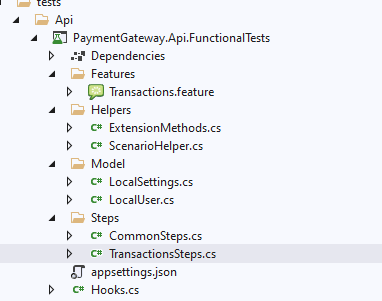


All the projects in UnitTests folder

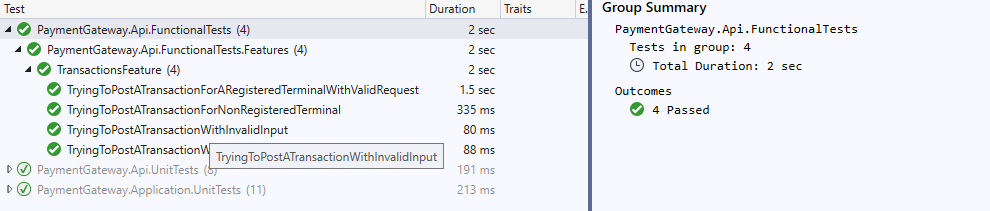
1. Uses NUnit3 as Testing Framework
2. Uses Moq,AutoFixture for building objects and Mocks
3. Uses FluentAssertions for assertions

## Section 3.5: Acceptance Tests

The technical solution also contains Acceptance tests written in BDD/Gherkin style using SpecFlow. These tests target the running instance of the application and works with real data.



The project has one feature file which has 4 scenarios to cover both happy/negative scenarios. The tests can be extended to include more scenarios.



Unlike unit tests, acceptance tests require the API up and running. Currently the API is not deployed.

I have opened 2 instances of Visual Studio. One instance is used to setup API up and running. I have run the SpecFlow tests from the second instance against the running API. Also make sure, the Acquirer API is also up and running. This is a temporary solution just to quickly test the solution.

## Section 3.5: Acquirer API

I have created another API in .Net Core 3.1 to simulate communications between PaymentGateway API and Acquirer. The solution is placed in another folder “Acquirer”. To simulate various possible responses from the Acquirer, it reads the last 4 digits of the Card and responds with a different result

|  |  |
| --- | --- |
| Last 4 digits of card | response |
| 1111 | Succeeded |
| 2222 | Failed due to InvalidCardDetails |
| 3333 | Failed due to LinkFailure |
| 4444 | Failed due to SomethingWentWrong.TryAgain |
| 5555 | Failed due to CancelledTryAgain |
| 6666 | Null. Error communicating with Acquirer |
| Any other card | Succeeded |

# Section 4: API Client Postman Collection & Swagger

Attached to the GitHub are the Postman collection scripts to test the two endpoints available.

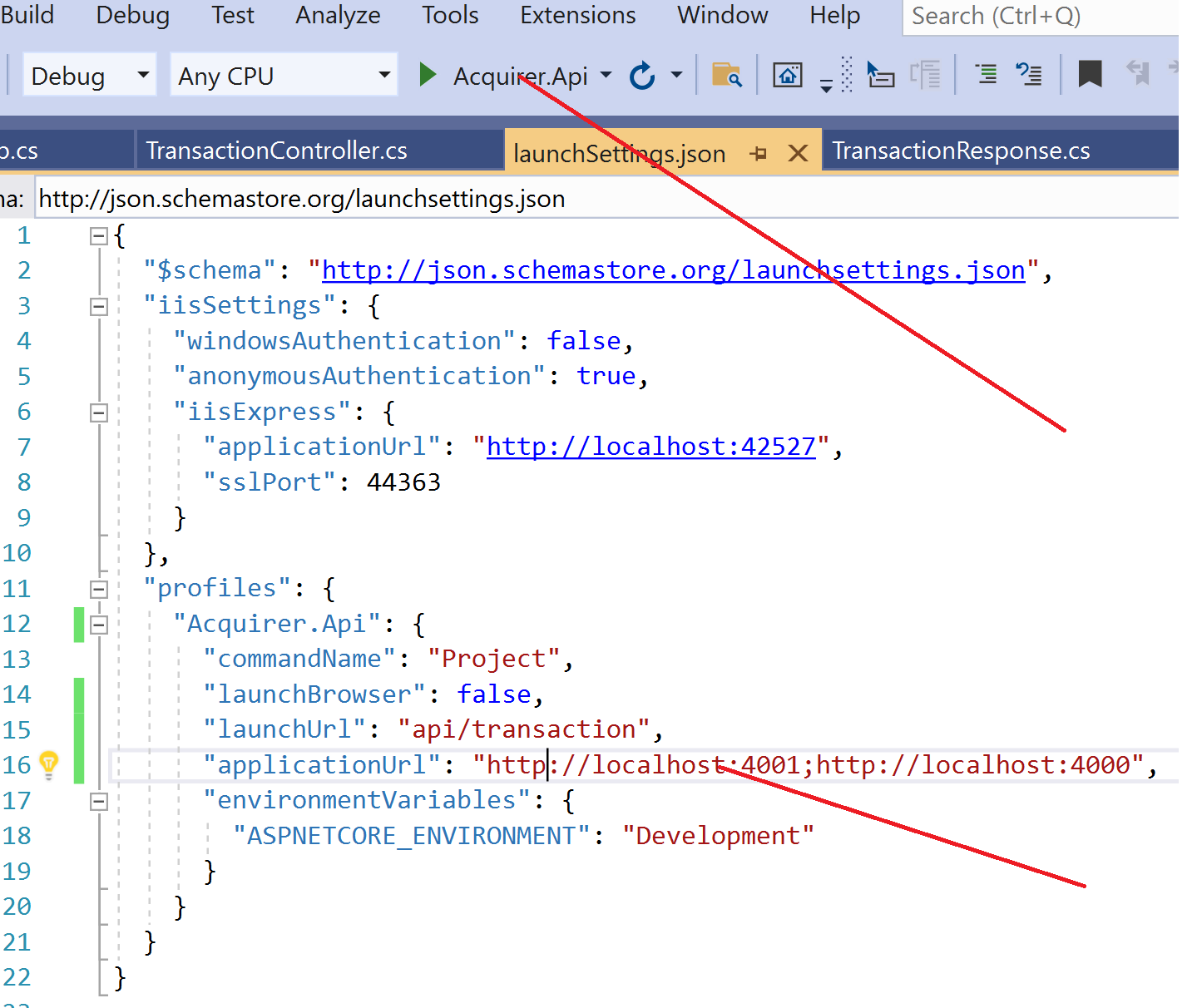
### 4.1 How to run PaymentGateway API up and running to send the requests through Postman

Follow the instructions below to set the API up and running.

#### 4.1.1 Setup Acquirer API up and running.

The PaymentGateway API depends on Acquirer API to send transactions and receive responses as described above.

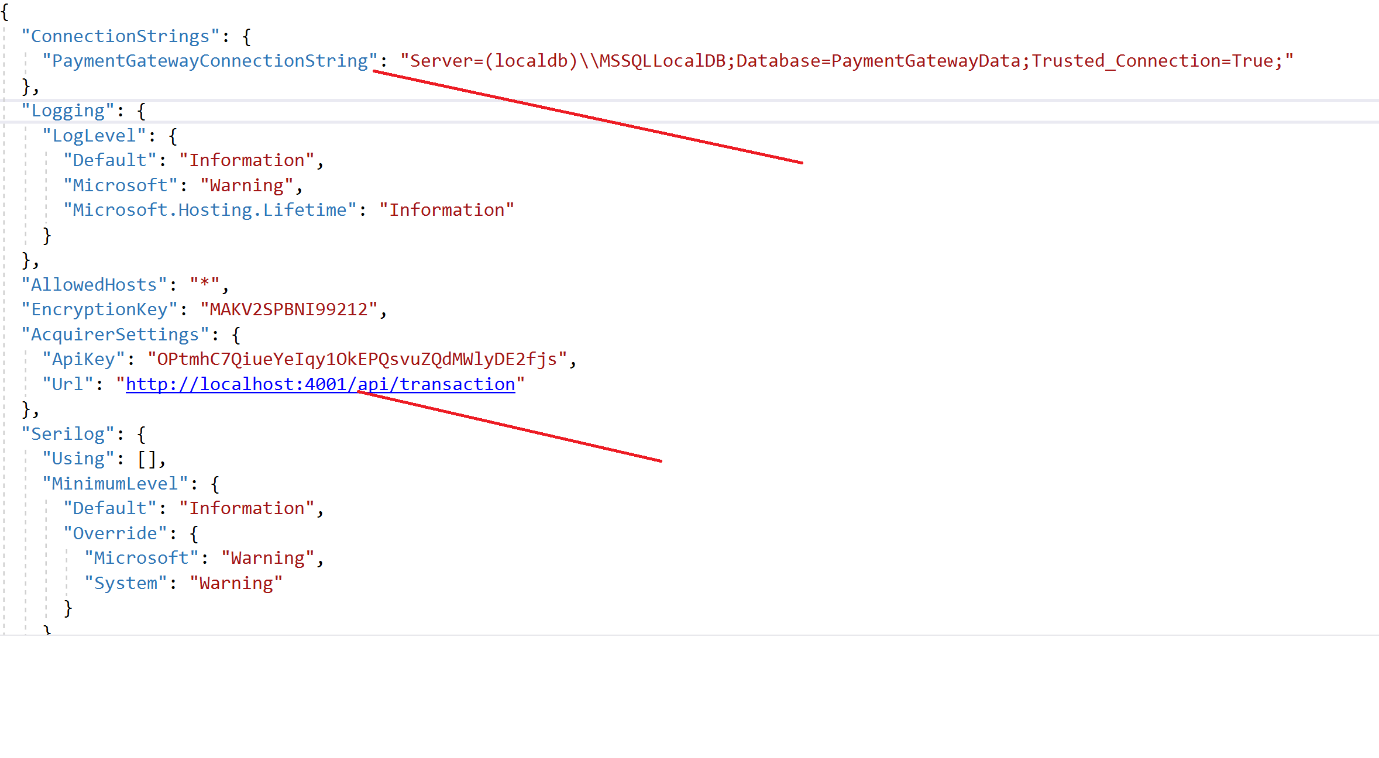
1. Open the Acquirer API Solution “Acquirer.sln” from Acquirer folder in Visual Studio 2019.

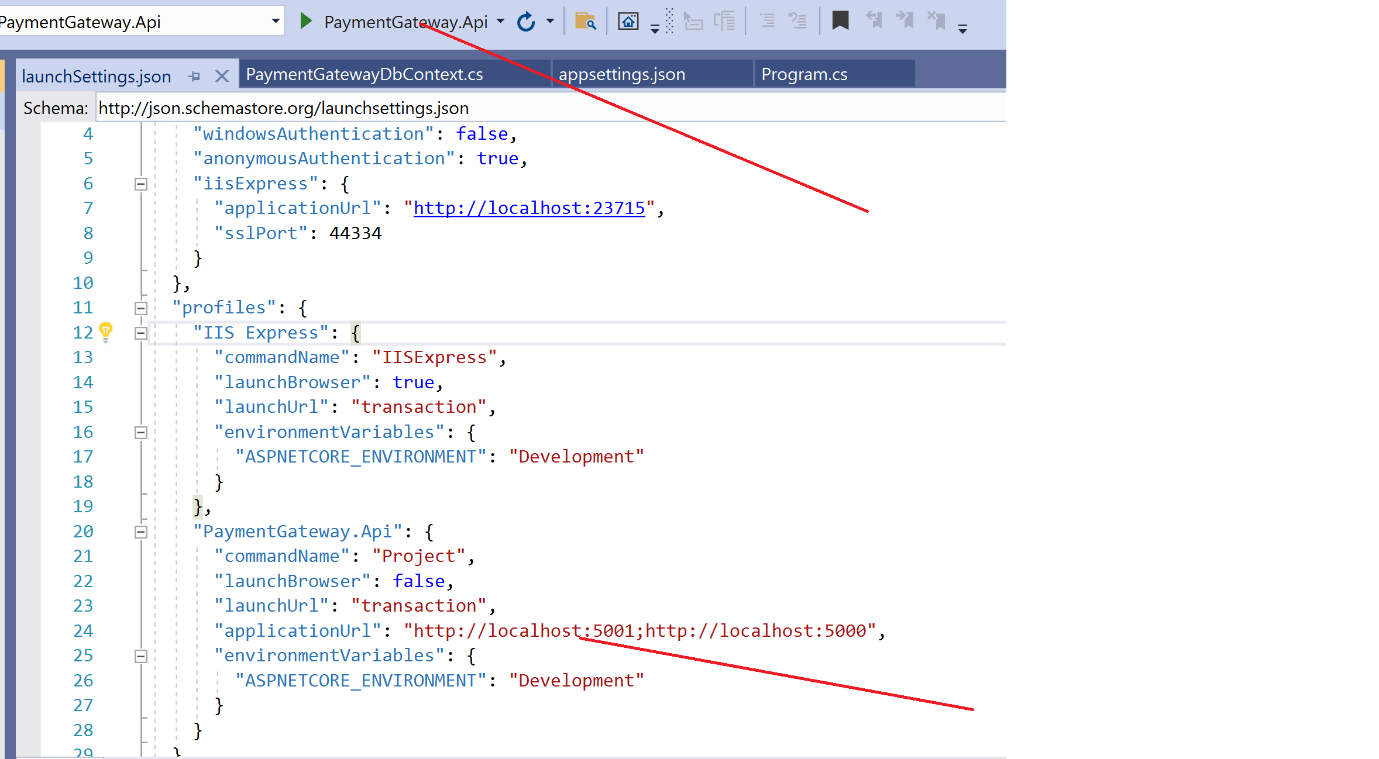


1. Point to the Acquirer.Api profile and launch the application. The “launchBrowser” is set to false. The application is launched in background and is available at <http://localhost:4001>

#### 4.1.2 Setup PaymentGateway API up and running.

1. Open the PaymentGateway.sln from PaymentGateway folder in VS2019.

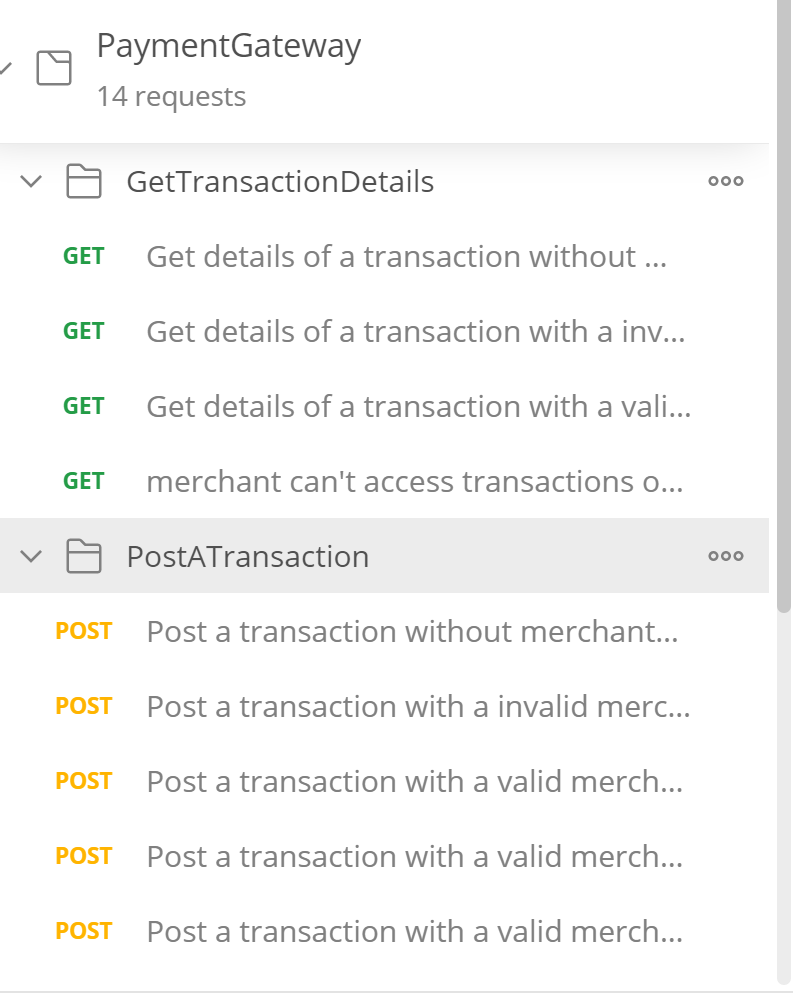
2. Make sure ConnectionString points to localdb. And AcquirerSettings.Url has correct url that Acquirer has been launched.3. 

3.Launch the application with selected profile. The PaymentGateway API will be up and running at <http://localhost:5001>. The Postman collection scripts are using this url. If you happen to change the url in the API solution, please remember to change in Postman collection.

#### 4.1.3 Postman Collections.

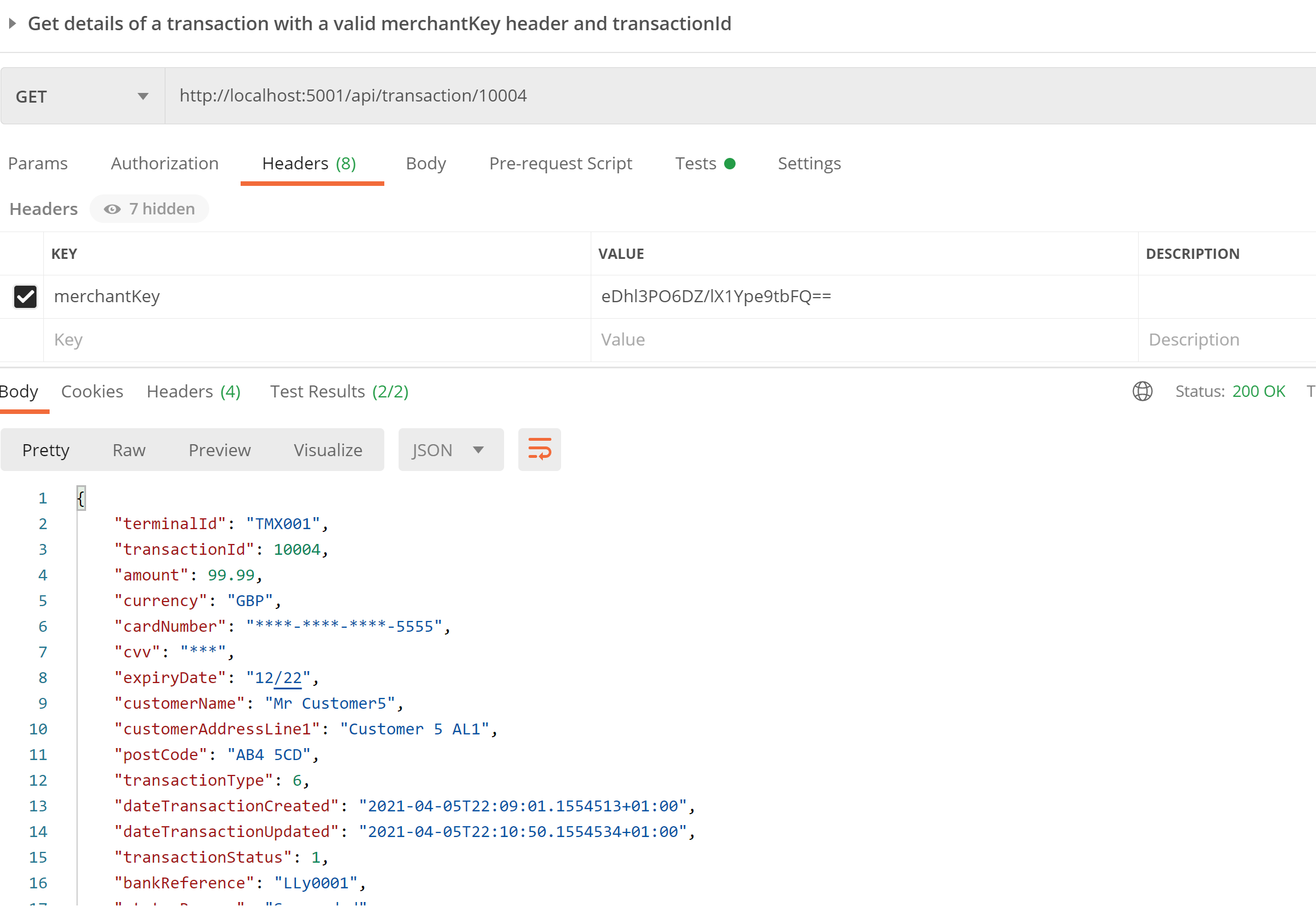
Postman collections are placed in “PaymentGateway.postman\_collection” file inside PaymentGateway solution folder. Import the collection into Postman

There are 2 folder, one for Get and the other for Post. There are various scripts to test both happy and negative tests.



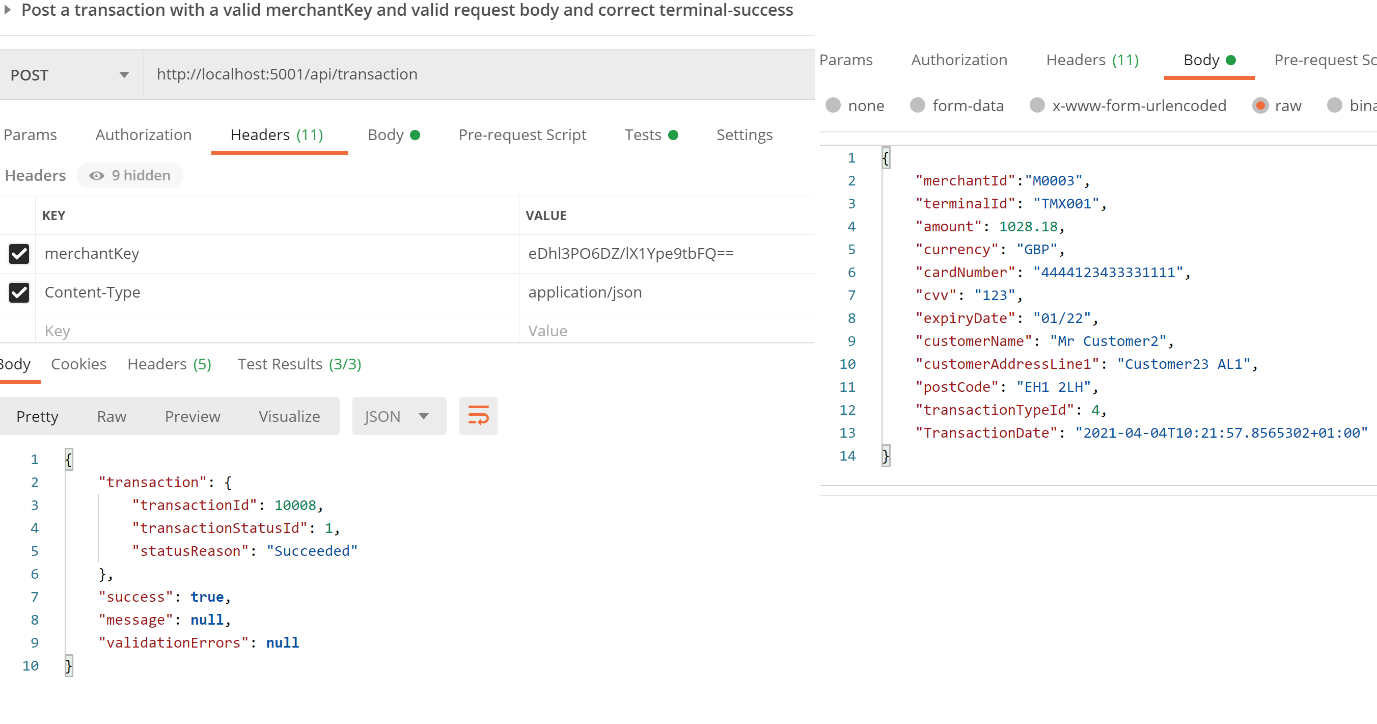
#### 4.1.4 Sample Valid Get request

to get transaction with id 10004 that belongs to MerchantId:M0003. The header key passed in the Header belongs to MerchantId:M0003. For transactionStatus and transactionType descriptions, refer to the database tables above.



#### 4.1.5 Sample Valid Post request

To Post a transaction for MerchantId:M0003. Make sure the merchantKey header belongs to that of M0003 merchant



Payload sample:

{

    "merchantId":"M0003",

    "terminalId": "TMX001",

    "amount": 1028.18,

    "currency": "GBP",

    "cardNumber": "4444123433331111",

    "cvv": "123",

    "expiryDate": "01/22",

    "customerName": "Mr Customer2",

    "customerAddressLine1": "Customer23 AL1",

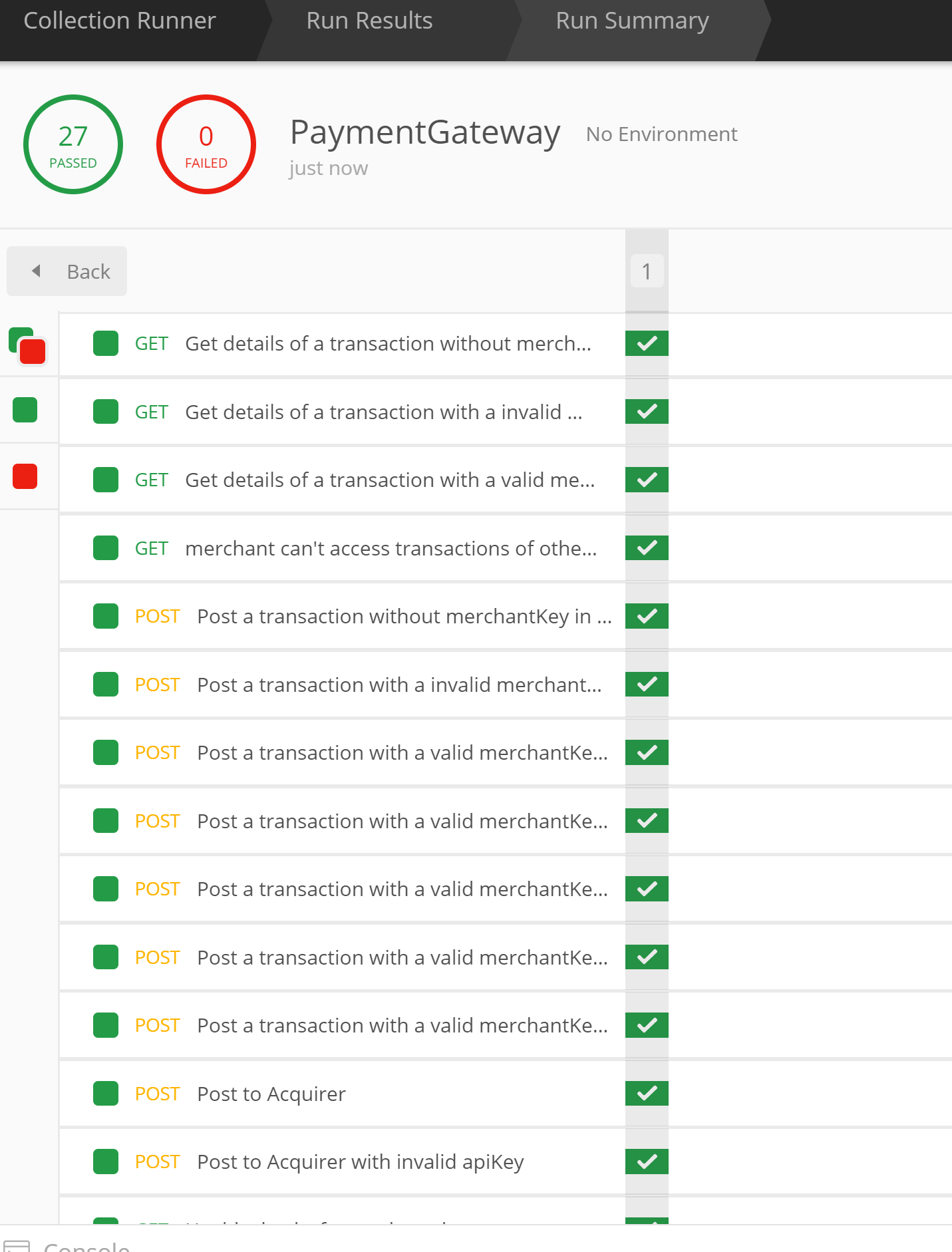
    "postCode": "EH1 2LH",

    "transactionTypeId": 4,

    "TransactionDate": "2021-04-04T10:21:57.8565302+01:00"

}

##### Collection runner results:



### 4.2 Swagger documentation & Swagger UI

#### 4.2.1 Swagger documentation

The Swagger documentation about the structure of the PaymentGateway API along with request types and response types can be found <http://localhost:5001/swagger/v1/swagger.json> while running the PaymentGateway API. This json can be used as input to any application that reads Open API specification to generate clients

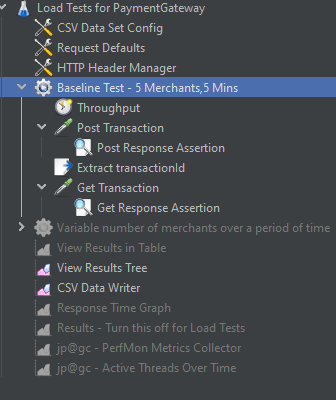
#### 4.2.2 Swagger UI

Swagger UI is accessible at [Swagger UI](http://localhost:5001/swagger/index.html) while running the PaymentGateway API.

# Section 5: Performance Testing JMeter

Attached to the Github is the Performance Test plan developed in JMeter. The test is based on the three merchants we have in the database and their terminals. The test plan has two thread groups.

I will introduce to the first one in this documentation which is Baseline Test - 1 TPS, 1 Thr, 5 Min

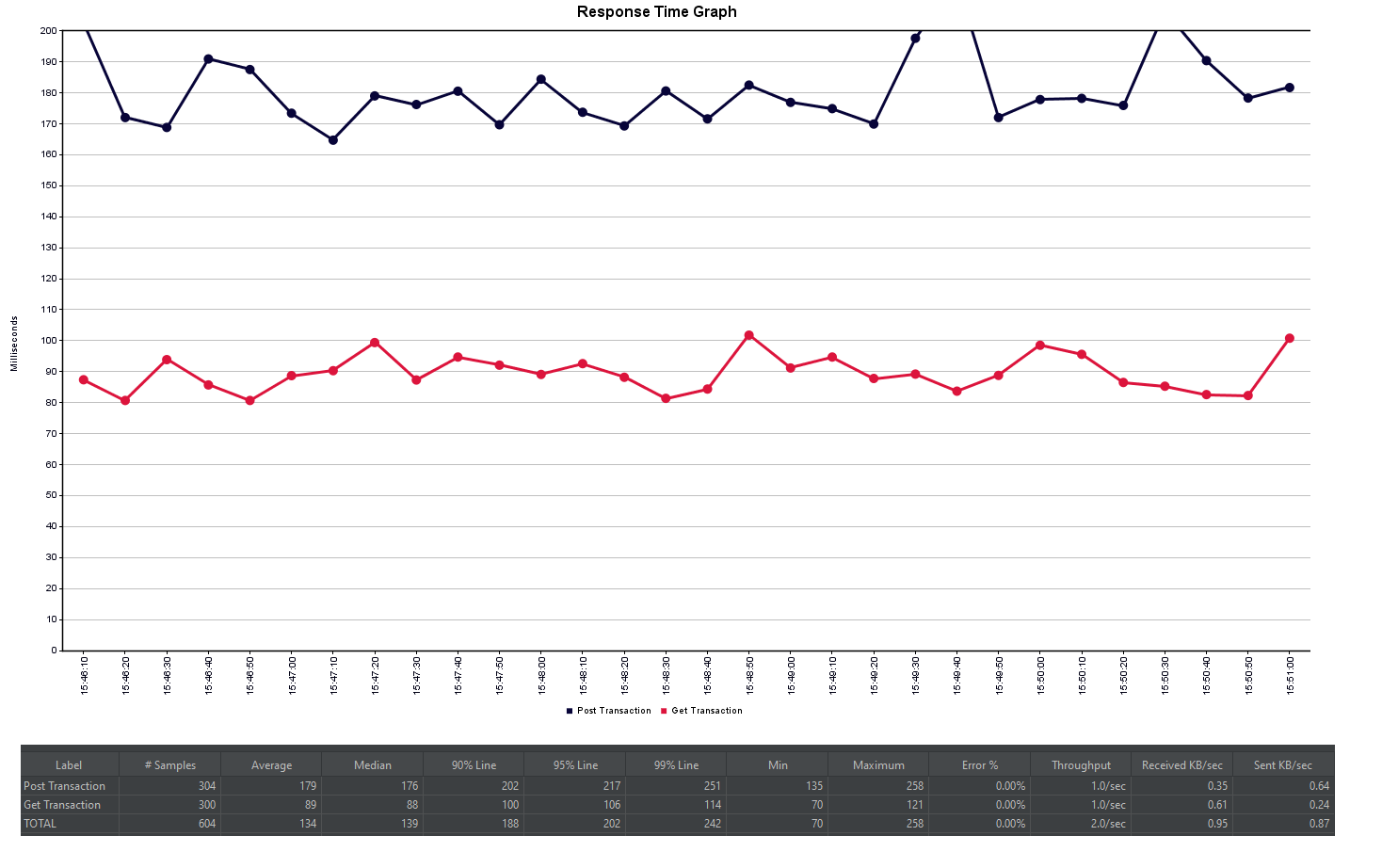


This thread simulates 5 concurrent merchants hitting the PaymentGateway API for 5 Mins. In each iteration.

1. Thread reads data from a CSV file, like MerchantId,TerminalId,MerchantKey,CardNumber
2. Post a transaction
3. Get the transaction details that has been created in step 2

This gives us to gauge the baseline performance of the application

Sample Results



## 5.1 Setting up JMeter and running the test

1. Install latest version of JAVA from <https://www.java.com/en/download/manual.jsp>

2. Install JMeter from <http://jmeter.apache.org/download_jmeter.cgi#binaries> . Download the binary [apache-jmeter-5.4.1.zip](https://mirrors.ukfast.co.uk/sites/ftp.apache.org/jmeter/binaries/apache-jmeter-5.4.1.zip) and extract to a local folder, for example C:\ [apache-jmeter-5.4.1](https://mirrors.ukfast.co.uk/sites/ftp.apache.org/jmeter/binaries/apache-jmeter-5.4.1.zip)\

3. Download [**plugins-manager.jar**](https://jmeter-plugins.org/get/) and put it into lib/ext directory, then restart JMeter.

4. Goto JMeter install location, for example C:\apache-jmeter-5.4.1\apache-jmeter-5.4.1\bin. And open JMeter.bat file. This open JMeter UI

5. From File Open, open the test plan file Load Tests for PaymentGateway.jmx.

6. To Start the tests, key Ctrl+R, or from menun Run->Start

7. After the test run is finished, results can be viewed using following listeners

7.1 View Results in Table

7.2 View Results Tree

7.3 Response Time Graph

7.4 Results

# Section 6: Further improvements

Improvements.

The following improvement can be implemented in future release.

### 5.1. Furthe endpoints

5.1.1 Ability to see all transactions for a merchant filter by date range, transaction type, status type, cardnumber etc

5.1.2 End of day batch report

5.2. The solution should have the abilities to detect duplicate transactions (based on card number, expiry date, TransactionDateTimeStamp, MerchantId, TerminalID)

5.3. The solution should be reliable. As soon as the transaction hits the PaymentGateway, store the transaction in the database and return a reference (transactionId) to the merchant. Then call the acquirer. In case if timeout occurs between merchant to PaymentGateway link while waiting for the response from Acquirer, the merchant has transactionId which they can use to query the status of the transaction before submitting a duplicate.

5.4. To make the PaymentGateway extensible, implement a authentication mechanism with username/password and JWT Token to store the claims of merchants . For example what transaction types the merchant can initiate or what terminals or payment amount limits they have. This gives a way to implement different subscription models based on merchant needs. For example to be able process AMEX cards, they should be subscribed to a higher plan.

5.5. Provide an option to those merchants who are willing to take risk. they can process the transactions in offline mode (for example when acquirer link is down). The technical solution should store the transaction locally and forward to Acquirer when Acquirer is up and running.

5.6. Use Industry standard encryption algorithms to store creditcard related data for PCI compliance.

5.7. Add retry logic to database access