PaymentAPI

> A set of HTTP RESTful endpoints to manage payment requests

Table of contents

General info

The solution provides with a payment system that allows a user to manage payment requests. The user can perform payment management operations linked to an Account with a monetary Balance in the system.

The user can perform the following operations:

- Create a Payment Request
- Cancel an existing Payment
- Process an existing Payment
- Retrieve an Account Balance

In addition to the requirements for the coding exercise I added the following functionalities which I feel give more flexibility to the user in order to test the functionalities implemented

- Create an Account
- Create one or more Deposit transactions for a give Account

Note: For convenience, as part of the database migration, an initial Account is created (with Id = 1) and an initial Deposit transaction is created so that the Opening Balance for that account is $100,000.00.

Solution Details

The solution **PaymentApiSolution** has been built as a Visual Studio Solution with a set of projects in C# including the technologies listed below.

It consists of four projects:

- PaymentApi.Api (ASP.Net MVC Core 3.1 Web API)
- PaymentApi.DataAccess (ASP.Net Core 3.1 Class Library)
- PaymentApi.Models (ASP.Net Core 3.1 Class Library)
- PaymentApi.XUnitTests (ASP.Net Core 3.1 Test Project)

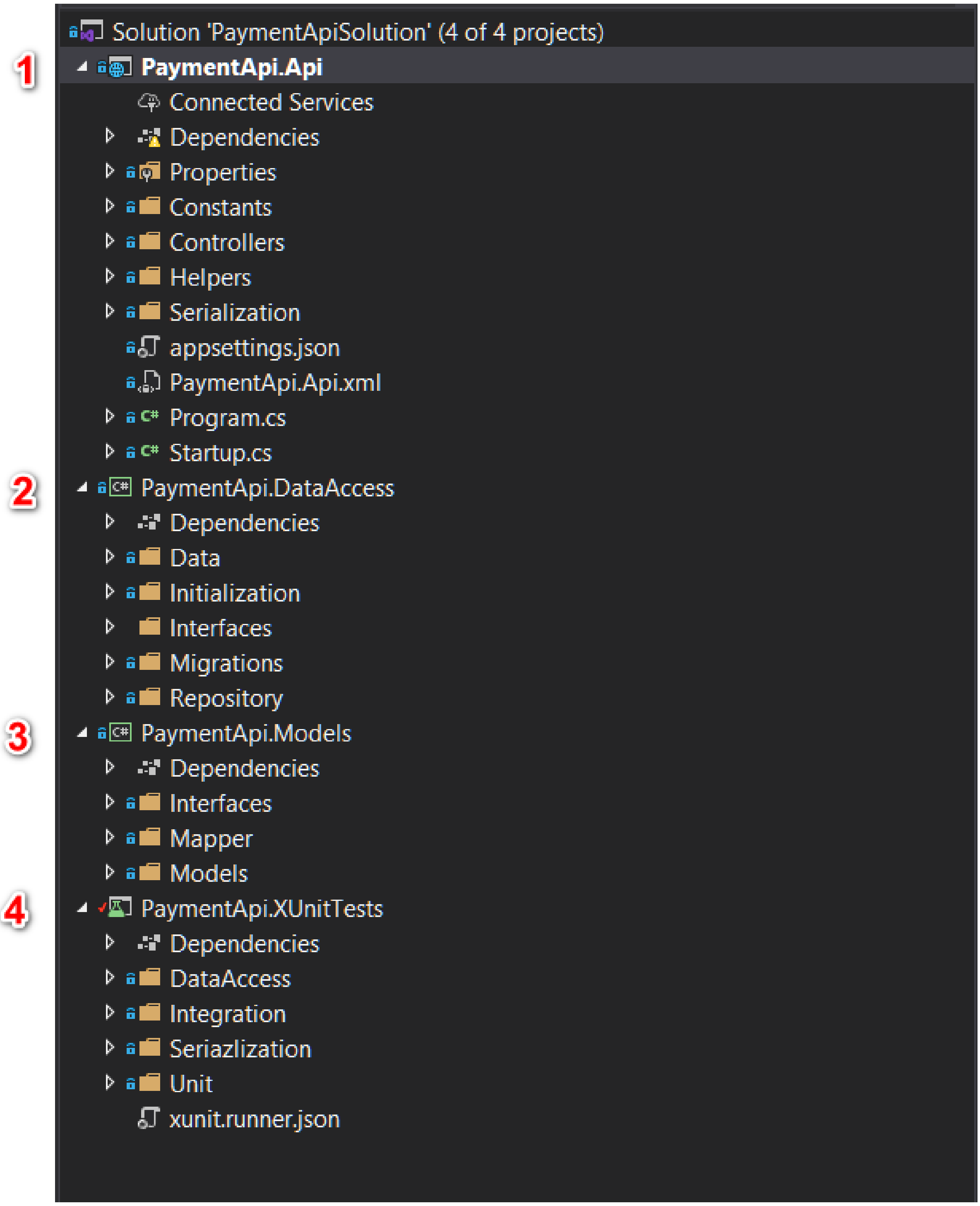
I decide to split up the solution in separate projects to have a clear separation of concerns between **API endpoints**, **Data Access** and **Model**.

In the Data Access layer I opted to use the Generic Repository pattern so that, if required, other Repositories can be added with a minimal effort as the base Repository class implements the minimum set of functionalities to perform CRUD operations on the database.

I chose Entity Framework Core as an ORM as I am more familiar with that and because being an ORM it offers out of the box an abstraction layer on top of the actual persistence layer. In this case I chose Microsoft SQL Server as storage as even if minimum, there is a relational pattern between Accounts and Transactions (even in a small scale application like this), because of my familiarity and experience with SQL Server and that fact that it is easily configurable when deploying to Azure from Visual Studio.

All Models, DTOs and AutoMapper profile configuration are in a separate class library for separation of concerns and maintainability.

For testing I used XUnit on an In-Memory database. The performances of In-Memory database outweigh the limitations of In-Memory SQL Lite database in Entity Framework as advanced features like Transactions are not required in my tests.



```
Solution 'PaymentApiSolution' (4 of 4 projects)
  PaymentApi.Api
      Connected Services
      Dependencies
      Properties
      Constants
      Controllers
      Helpers
      Serialization
      appsettings.json
      PaymentApi.Api.xml
      Program.cs
      Startup.cs
  PaymentApi.DataAccess
      Dependencies
      Data
      Initialization
      Interfaces
      Migrations
      Repository
  PaymentApi.Models
      Dependencies
      Interfaces
      Mapper
      Models
  PaymentApi.XUnitTests
      Dependencies
      DataAccess
      Integration
      Seriazlization
      Unit
      xunit.runner.json
```

PaymentApi.Api

This project is the Web Api project type exposing the HTTP endpoints and is configured with Swagger so that the user can also see the available endpoints and play with them directly in SwaggerUI without the need to use an external tool if he/she prefers.

PaymentApi.DataAccess

This project is a Class Library that is the center of Entity Framework Database configuration and Migrations and also the data Repositories

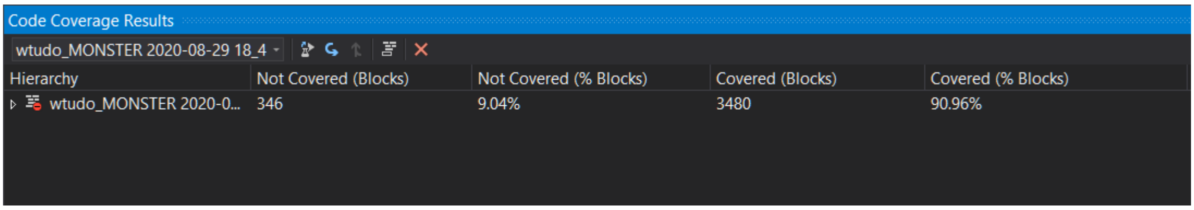
PaymentApi.Models

This project is a Class Library that contains all Models, DTO classes and Automapper profile configuration.

PaymentApi.XUnitTests

This is the testing project. Tests have been built using xUnit and FluentAssertions.

Test Coverage registered using Visual Studio 2019 Enterprise edition recorded the following:



Code Coverage Results

wtudo_MONSTER 2020-08-29 18_4

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
wtudo_MONSTER 2020-0...	346	9.04%	3480	90.96%

Technologies

- ASP.Net Core 3.1 Web Api
- Entity Framework Core
- Serilog
- Swagger, SwaggerUI, Swashbucket
- SQL Server
- InMemory Database
- XUnit
- Automapper
- FluentAssertions
- Newtonsoft

Setup

Prerequisites:

- DotNet Core 3.1 installed on testing/building machine.
- Microsoft SQL Server (any edition)

The solution is provided in a zip file. Unzip the file content in a folder (e.g. PaymentAPI)

Connection String

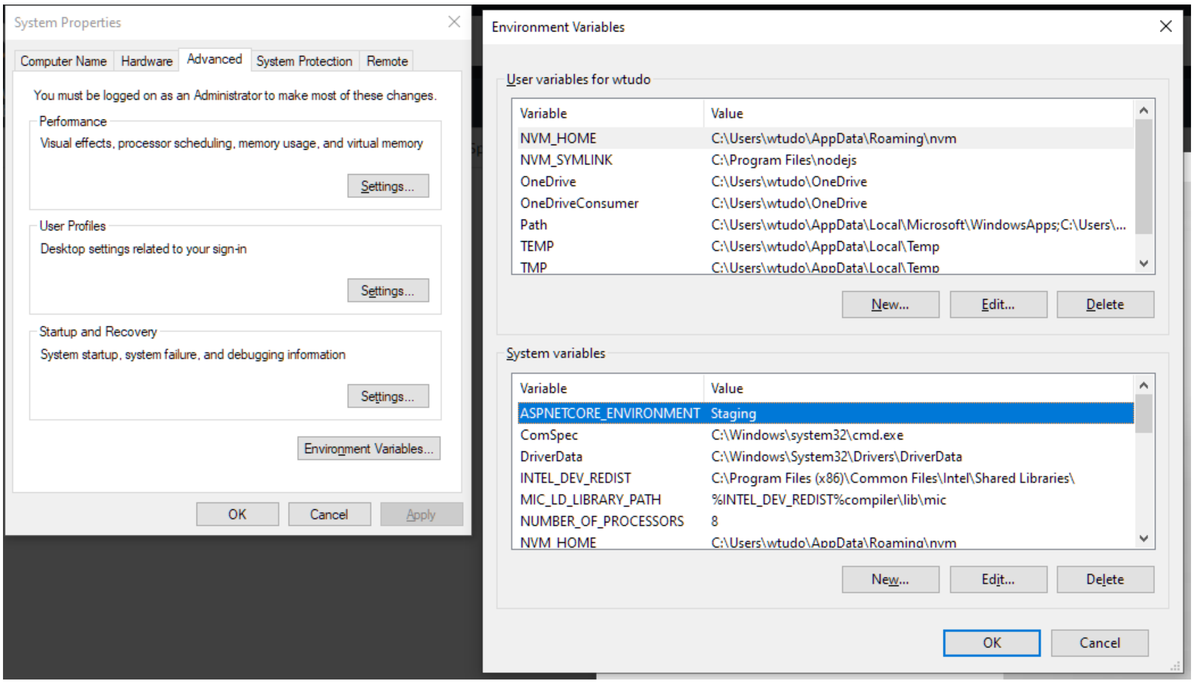
Modify the configuration file in

```
<Solution Folder>\PaymentApi.Api\appsettings.json
```

Environment Variables

and change the details in the connection strings for the three environments (**Development, Staging and Production**).
Add or modify your system environment variable **ASPNETCORE_ENVIRONMENT** to point to one of the three environments (**Development**, **Staging** or **Production**.)

You can accomplish using Windows Environment Variables



You must be logged on as an Administrator to make most of these changes.

Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your sign-in

System startup, system failure, and debugging information

User variables for wtudo

Variable	Value
NVM_HOME	C:\Users\wtudo\AppData\Roaming\nvm
NVM_SYMLINK	C:\Program Files\nodejs
OneDrive	C:\Users\wtudo\OneDrive
OneDriveConsumer	C:\Users\wtudo\OneDrive
Path	C:\Users\wtudo\AppData\Local\Microsoft\WindowsApps;C:\Users\...
TEMP	C:\Users\wtudo\AppData\Local\Temp
TMP	C:\Users\wtudo\AppData\Local\Temp

System variables

Variable	Value
ASPNETCORE_ENVIRONMENT	Staging
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
INTEL_DEV_REDIST	C:\Program Files (x86)\Common Files\Intel\Shared Libraries\
MIC_LD_LIBRARY_PATH	%INTEL_DEV_REDIST%compiler\lib\mic
NUMBER_OF_PROCESSORS	8
NVM_HOME	C:\Users\wtudo\AppData\Roaming\nvm

or you can use command line command (Windows **cmd** or **powershell** to set the environment value at System level line as well. e.g. for "Development"

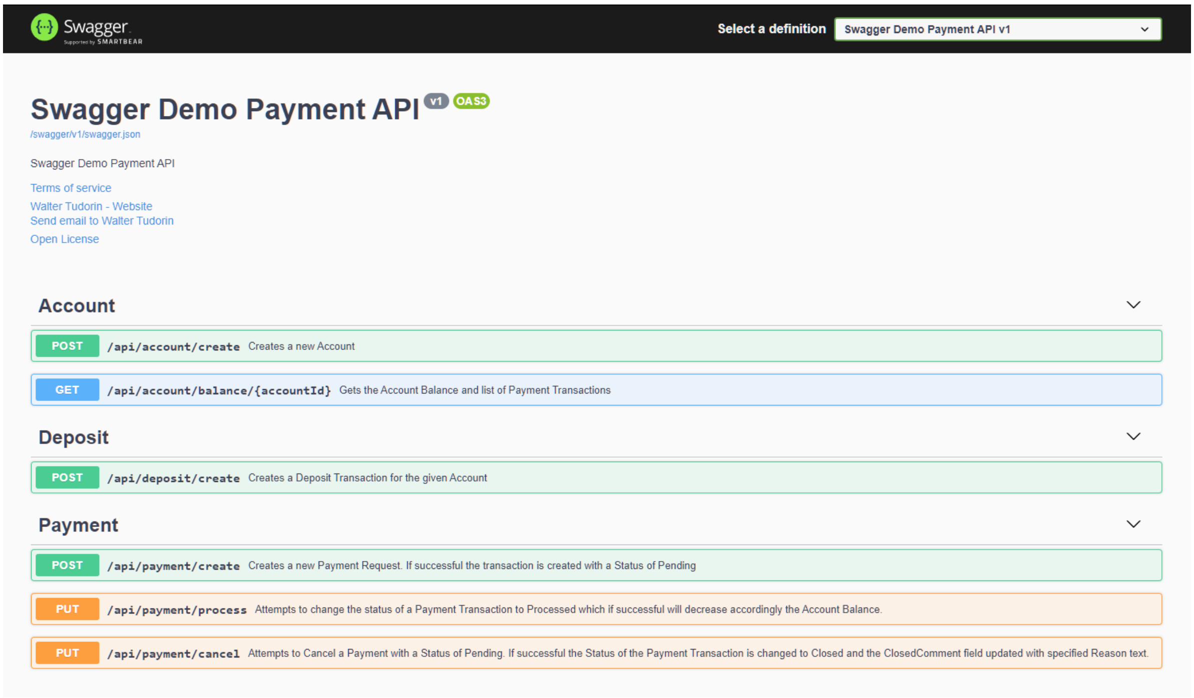
```
CMD: setx ASPNETCORE_ENVIRONMENT "Development" /M   (Requires Administrative
rights)

Powershell: [Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT",
"Development", "Machine")
```

Build, Run and Test

Visual Studio

Open the solution and make sure that the project **PaymentApi.Api** is the active project in the solution. Press F5.

The active project is **PaymentApi.Api** which has been configured to display Swagger UI in the browser.



Select a definition: Swagger Demo Payment API v1

Swagger Demo Payment API v1 OAS3

/swagger/v1/swagger.json

Swagger Demo Payment API

Terms of service
Walter Tudorin - Website
Send email to Walter Tudorin
Open License

Account

POST /api/account/create Creates a new Account

GET /api/account/balance/{accountId} Gets the Account Balance and list of Payment Transactions

Deposit

POST /api/deposit/create Creates a Deposit Transaction for the given Account

Payment

POST /api/payment/create Creates a new Payment Request. If successful the transaction is created with a Status of Pending

PUT /api/payment/process Attempts to change the status of a Payment Transaction to Processed which if successful will decrease accordingly the Account Balance.

PUT /api/payment/cancel Attempts to Cancel a Payment with a Status of Pending. If successful the Status of the Payment Transaction is changed to Closed and the ClosedComment field updated with specified Reason text.

Tu run the xUnit tests simply open **Test Explorer** and run all.

If you have the Enterprise version of Visual Studio you can also check the % of Code Coverage.

Command Line

If you prefer you can use **DotNet CLI**

On windows

Build Solution

```
cd <solution folder>
dotnet build PaymentApiSolution.sln
```

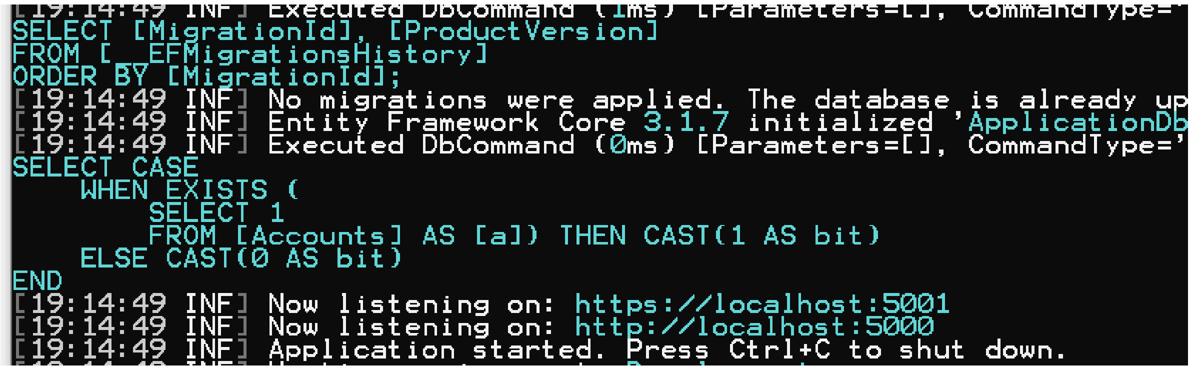
Run Tests

```
<in solution folder>
dotnet test
```

Run Web Api Project

```
<in solution folder>
cd cd PaymentApi.Api
dotnet run
```

If successful Kestrel web server will be listening on port 5000 for HTTP and 5001 for HTTPS.



```
[19:14:49 INF] Executed DbCommand (1ms) [Parameters=[], CommandType='
SELECT [MigrationId], [ProductVersion]
FROM [__EFMigrationsHistory]
ORDER BY [MigrationId];
[19:14:49 INF] No migrations were applied. The database is already up
[19:14:49 INF] Entity Framework Core 3.1.7 initialized 'ApplicationDb
[19:14:49 INF] Executed DbCommand (0ms) [Parameters=[], CommandType='
SELECT CASE
    WHEN EXISTS (
        SELECT 1
        FROM [Accounts] AS [a]) THEN CAST(1 AS bit)
    ELSE CAST(0 AS bit)
END
[19:14:49 INF] Now listening on: https://localhost:5001
[19:14:49 INF] Now listening on: http://localhost:5000
[19:14:49 INF] Application started. Press Ctrl+C to shut down.
```

If you browse to http://localhost:5000/index.html or https://localhost:5001/index.html you will see the same SwaggerUI as in the image above.

Demo on Microsoft Azure

This solution has been deployed to Azure and you can play with that browsing to https://wt001-paymentapiapp-aps.azurewebsites.net/index.html

Payment API Endpoints

For an HTML static version of the description of each end point, input and output see html file in

`<Solution Folder>\README.Media\Payment API.html`

Notes on actual Implementation Vs Requirements

Assumptions & Terminology

- ***Opening Balance***: the sum of all amounts of Deposit Transactions (type Deposit and a status of Processed).
- ***Pending Balance***: the sum of all amounts of Pending Payment Transactions (type Withdrawal and a status of Pending).
- ***Processed Balance***: the sum of all amounts of Processed Payment Transactions (type Withdrawal and a status of Processed).
- ***Closing Balance***: calculated as **Opening Balance** (minus) **Pending Balance** (minus) **Processed Balance**.

Important Note: At this stage the Account Balance is not a balance calculated at a specific point in time, that takes in consideration Deposits and Payments dates, but purely a mathematical calculation of balances, regardless of the related transaction Date.

1. As a user I want to create a payment request. Required fields: amount, date The customer balance must be enough for the amount provided, if not the request is created and closed immediately with a closed comment "Not enough funds" If the request is successful the payment is considered pending Validation errors are shown for missing fields or wrong values I can see the new payment request in the list of payments marked as "Pending" (see user story no 4)

See endpoint **/api/payment/create**

Variations

I added the functionality for the User to create additional Accounts (other that the initial one seeded) and add monies to the Account's balance creating Deposit transactions. This allows for more flexible and creative testing.

See endpoints **/api/account/create** and **/api/deposit/create.**

2. As a user I want to cancel a payment request Fields (not required): reason (text) A payment that is processed can't be cancelled The payment is considered "Closed" with the above closed reason

See endpoint **/api/payment/cancel**

Variations

In the Json body, part of the request, along with the **TransactionId** used to identify the specific Transaction Payment to Cancelled I added the **AccountId** to double check that we are indeed cancelling a payment related to the correct Account.

3. As a user I want to process a payment request A payment that is closed can't be processed The payment is considered "Processed"

The balance is reduced by the payment amount

Variations

In the Json body, part of the request, along with the **TransactionId** used to identify the specific Transaction Payment to Cancelled I added the **AccountId** to double check that we are indeed processing a payment related to the correct Account.

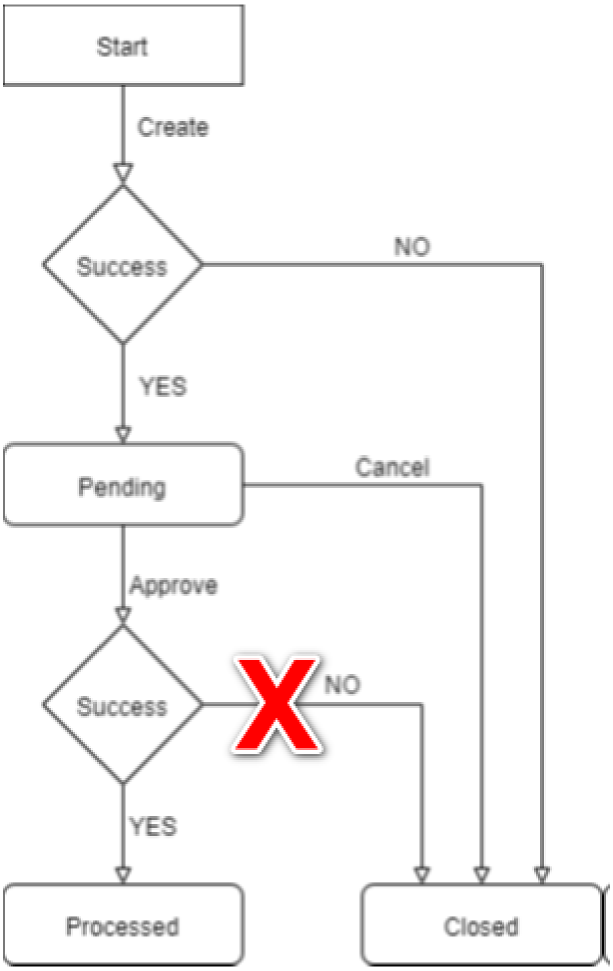
Note As part of the functionality to "Process" a payment the requirements implemented assume that the payment status is

1. Not Closed
2. Not Processed
3. Is Pending

There are no other consideration performed before processing a Payment with status of Pending as the Balance check would have been performed before creating the Payment.

Given this assumption the decisional branch in the flow chart provided marked with a red X has not been implemented as the only possible scenarios are:

- A Payment has a status of "Pending" -> (Is then Processed) -> Status goes to "Processed" (YES flow direction)
- A Payment has already a status of "Processed" -> (no change of status occurs) -> Status remains "Processed"
- A Payment is Closed -> (no change of status occurs) -> Status remains "Closed"



4. As a user I want to see my balance and a list of payments. Fields: account balance and for the payment list: date, amount, status Closed reason if it exists Sorted by newest date

Variations

In the response Json body, I am returning additional information such as

- AccountId
- Opening Balance amount
- Processed Balance amount
- Pending Balance amount
- Closing Balance amount
- List of Deposit Transactions
  - Id

    - AccountId
    - Amount
    - Date

- List of Payment Transactions sorted by Date Descending (most recent payments first)
  - Id
    - AccountId
    - Amount
    - Date
    - TransactionStatus
    - Closed Reason

Configurations and Environments

The application can be configured for three different environments

- **Development**
- **Staging**
- **Production**

At this stage the only diversification is in regards to the Database the application will be connected to. See Environment variables in the Setup section.

Testing

The tests implemented cover Integration Testing where we call API endpoints and Unit Test of Helper classes using an In-Memory database configured in the PaymentApi.APi StartUp class. Test coverage in my tests reaches 90+%. Logging in tests is not required so ILogger has been mocked.

Future Improvements / Extensions

Being a coding exercise but with the assumption it would be deployed in a production environment the following is a list of feature/ extensions/ additional functionalities that could be implemented on top of the current code base.

- Authentication. Authentication is missing and should be added.
- Support for Balances calculated at a specific point in time could be added.
- Structural Logging has been add using SeriLog. A complete integration configuration for SEQ or Azure Application Insight is outside of the scope of this codebase but should be added depending on where the application is eventually going to be deployed.
- More granular and specific logging if required.
- Throttle incoming requests with tools like Azure API Management.

CI / CD

My experience as a developer around Continuous Integration and Continuous Delivery is based on the usage of tools like JIRA for job tracking and integration with BitBucket on-premises and Bamboo for automated Build, Testing and Deployments, as an end user and not specifically configuring BitBucket Build Plans or Bamboo. As such I cannot provide with deployment scripts but just comment on the topic.

The Application built should be integrated into the company's CD/CD platform with a set of tools like JIRA, BitBucket and Bamboo where

- JIRA allows for Project and Issue Tracking.
- BitBucket for code management giving teams one place to plan projects, collaborate on code, test, and deploy.
- Bambooo for continuous integration, deployment, and delivery