# **Building a Payment Gateway API**

E-commerce is currently experiencing exponential growth, and with it comes the increasing need for merchants to sell their goods and services online. These merchants require a seamless way to collect payments from their customers. To cater to this demand, we intend to develop a robust payment gateway - an API-based application that empowers merchants to provide their shoppers with a convenient method for paying for their purchases.

# **Requirements**

The product requirements for this initial phase are the following:

* A merchant should be able to process a payment through the payment gateway and receive either
* a successful or unsuccessful response.
* A merchant should be able to retrieve the details of a previously made payment. The next section

## **Retrieving a payment’s details**

The second requirement for the payment gateway is to allow a merchant to retrieve details of a

previously made payment using its identifier. Doing this will help the merchant with their reconciliation and reporting needs. The response should include a masked card number and card details along with a status code which indicates the result of the payment.

# **Deliverables:**

**Develop an API that enables merchants to:**

* Initiate payment processing through the payment gateway.
* Retrieve details of previously completed transactions.

Create a mock bank simulator to facilitate thorough testing of the payment gateway API.

# **Assumptions:**

* The requirement is to mask card details during the retrieval of previously completed transactions, and I have implemented data masking while storing the data to comply with data protection regulations.
* CVV codes will consist of either 3 or 4 digits.
* Card numbers will fall within the range of 13 to 19 digits.
* Foreign exchange conversion will not be required initially, as the system will exclusively support GBP transactions.
* Previous payment details by Merchants not by paymentId

# **Project Architecture:**

I have implemented the CQRS and MediatR patterns, adhering to SOLID design principles to ensure a robust and flexible architecture.

# **Project solution structure:**

The project's structure will be based on the Clean Architecture principles outlined in Uncle Bob's blog (<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>).

* **Paymentgateway.API:** This layer houses the API controllers responsible for handling incoming HTTP requests and returning responses. It serves as the entry point for external clients to interact with the application.
* **Paymentgateway.Application:** This layer encapsulates the core business logic and rules. It contains interfaces that define the contracts for various application services. This layer is independent of external concerns and focuses on implementing the application's use cases.
* **Paymentgateway.Domain:** Here, you'll find the domain entities that represent the fundamental concepts of clean architecture. It also holds repository interfaces, which define how to access and manage data related to these entities. The domain layer is the heart of the application's business logic.
* **Paymentgateway.Persistence:** This layer is responsible for implementing the repository interfaces defined in the domain layer. It handles data storage and retrieval operations, connecting your application's business logic with the underlying database. It maintains separation from the domain layer to ensure a clear distinction between data manipulation and business rules.
* **PaymentGateway.Infrastructure:** This layer manages interactions with external services, such as third-party APIs. It ensures that these interactions are isolated from the core business logic and data layers, promoting modularity and ease of maintenance.

# **Pre-Requisites:**

* .NET 7
* Docker Desktop, if you would like to run the API on the docker container

# **How to run the application:**

**Visual Studio:**

* Open the solution file in Visual Studio.
* Press F5 or click the "Start" button to build and run the Web API application locally.
* Visual Studio will launch a local development server and open a web browser, allowing to test the API endpoints.

**Command Line:**

* Open a command prompt or terminal window.
* Navigate to the root directory "Paymentgateway.API" of the Web API project
* Use the .NET CLI to build and run the application:

**dotnet build**

**dotnet watch or dotnet run**

The Web API will be hosted locally and accessible at a URLs  [**http://localhost:5057/swagger/index.html**](http://localhost:5057/swagger/index.html) and [**https://localhost:7099/swagger/index.html**](https://localhost:7099/swagger/index.html)

**Docker:**

* Make sure you have Docker Desktop installed on your machine.
* Open a command prompt or terminal window.
* Navigate to the root "Paymentgateway.API" directory of the Web API project.
* Build a Docker image and run using the command:
* docker-compose up

The Web API should now be accessible at http://localhost:8080.

# **Additional Features:**

In addition to the core functionality, I have added the following features:

* API Key Authentication for application.
* Structured Logging using Serilog for enhanced traceability.
* Segregated services for Acquiring Banks, Input & Card Validation, and Data Masking.
* Utilization of Docker containers for efficient deployment and scalability.

# **Further Improvements:**

* **Security**: I have added API key authentication for application authentication, focusing on security beyond user authentication. Need to consider implementing Authentication, Authorization, and Multi-Factor Authentication, alongside Rate Limiting.
* **Tests**: Due to time constraints, I have currently included a limited number of tests. It is essential to allocate time for creating more comprehensive Unit and Integration test scenarios. This approach will ensure coverage of all possible scenarios and achieve maximum code coverage.
* **Communication**: Integration of Email and SMS notification services will enhance communication with users and merchants.
* **Reporting**: To provide valuable insights, a separate Reporting Service needs to be developed.
* **Scalability**: To ensure scalability and maintainability as the system grows, need to create distinct services for Banking, Validation, and Data Masking,
* **Resilience**: Need to include the Polly library for resilience.
* **Monitoring**: Application logs need to be sent to tools such as Cloudwatch and AppInsights for efficient monitoring. Moreover, need to integrate with API Metrics tools like DataDog and Grafana for in-depth performance analysis.
* **Database Optimization**: Consider transitioning data storage from SQLite to SQL or NoSQL databases for enhanced data management
* **CI/CD**

# **Cloud Technologies:**

My decision between these cloud platforms would be influenced by the potential for cost optimization.

## **Microsoft Azure:**

Azure is particularly advantageous for deploying .NET applications due to its tight integration with the Microsoft technology stack.

* **Azure App Service:** Hosting the Web API as an Azure App Service. It provides easy deployment, scaling, and management.
* **Azure Kubernetes Service (AKS):** For more control over containerized deployments, AKS is a managed Kubernetes service that can host the Web API in containers.
* **Azure Functions:** For smaller microservices or event-driven scenarios, we can use Azure Functions to deploy individual functions as services.
* **Azure SQL Database:** For databases, Azure offers SQL Database for managed relational databases and Cosmos DB for NoSQL databases.

## **Amazon Web Services (AWS):**

AWS is a widely used cloud platform with a variety of services for hosting .NET applications.

* **AWS Elastic Beanstalk:** Similar to Azure App Service, Elastic Beanstalk provides a platform for deploying and managing web applications.
* **Amazon ECS:** For containerized deployments, Amazon Elastic Container Service (ECS) or Amazon Elastic Kubernetes Service (EKS) can be used.
* **AWS Lambda:** Like Azure Functions, AWS Lambda enables serverless microservices.
* **Amazon RDS:** For managed relational databases, you can use Amazon RDS, and for NoSQL databases, consider Amazon DynamoDB.