

CARD COST API

January, 2024
Verónica Seoane

Index

Case Study	3
Business Considerations	6
Technical tools	7
Utilised libraries	7
Technical Decision	7
Areas for improvement	9
Instructions	11
To run local	11
To run with Docker	11

Case Study

CARD COST API

OVERVIEW

The Card Cost API is a part of the Etraveli Group hiring process, and will help us determine your level of competence. You are strongly encouraged to approach it with a professional attitude and deliver a working, production - grade solution.

Please keep in mind that this code will be reviewed and evaluated, and we expect you to be able to present it and probably even change it during your interview.

Below you will find the requirements of the challenge in full detail, as well as a comprehensive QA section, which you are urged to consult before contacting us with any questions. We will of course be available for any additional clarifications or questions that may arise.

We thank you in advance for your time and effort, and wish you good luck.

REQUIREMENTS

Background

A payment card number, primary account number (PAN), or simply a card number, is the card identifier found on payment cards.

The PAN is an 8 to 19 digit number displayed on one side of the card.

The first 6 digits of a payment card number (credit cards, debit cards, etc.) are known as the Issuer Identification Numbers (IIN), previously known as Bank Identification Number (BIN). These identify the institution that issued the card to the card holder.



Task

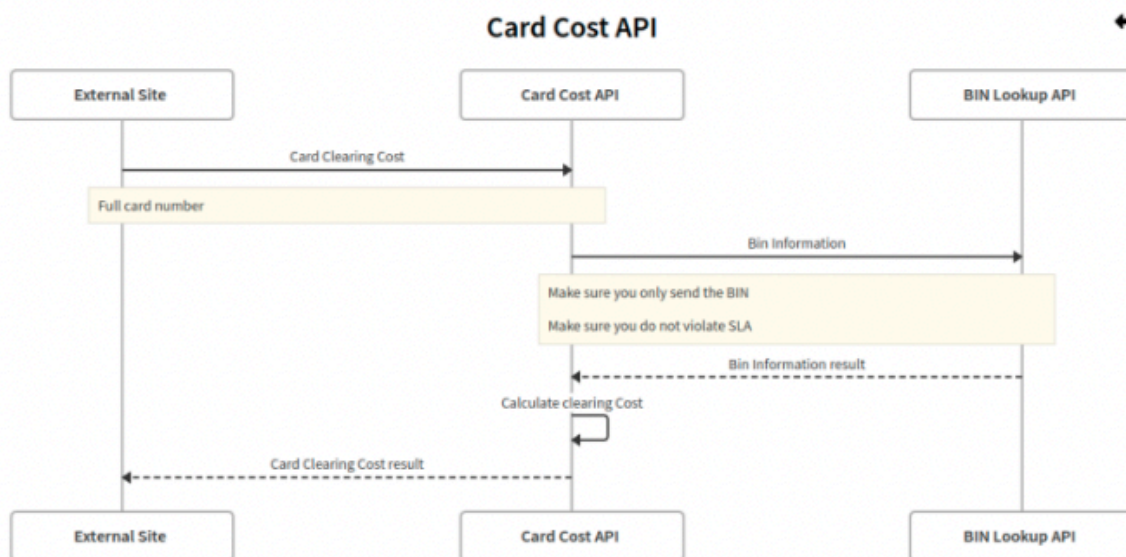
Given the following clearing cost matrix:

Card issuing country	Clearing Cost (USD)
US	\$5
GR	\$15
Others	\$10

We would like you to create a REST API that can do the following:

- Have full Create Read Update Delete operations on the clearing cost matrix table
- Can respond with the clearing cost of a given card number, utilizing the information provided by this public API <https://bintable.com/get-api>

The card clearing cost calculation is depicted in the following diagram:





Required deliverable traits

- A web (HTTP) API that can support the following action
 - HTTP POST on /payment-cards-cost of the following JSON:

```
{  
  "card_number": <pan>  
}
```

- Response:

```
{  
  "country": <iso2_code>,  
  "cost": <decimal>,  
}
```

- The implementation of the API should be written in JAVA
- Comprehensive unit and integration tests
- Full source code
- Comprehensive documentation

Bonus points

- Docker deployable solution
- Detailed instructions
- Security considerations
- High availability
- Extendability

QA

Q: How many calls per minute should I expect in my API ?

A: From 1 to 7000 per minute.

Q: What java framework should I use?

A: Anyone you like!

Q: I discovered a bug but have already submitted my code! What can I do?

A: Tell us! It is ok to re-submit your code within a reasonable amount of time.

Business Considerations

- In this project, it was used <https://binlist.net/> instead of <https://bintable.com/> due a issues related with API_KEY access.
- The first 6 or 8 digits of a card number are known as the Issuer Identification Number (IIN)^[1]. For this project, only the first 6 digits were considered, guided by the letter of the problem.
- This application allows you to receive the PAN (Primary Account Number) guided by the letter of the problem. In my opinion, a better approach would have been to receive only the IIN, for security purposes. Otherwise, I would request the external site to send to the system the PAN encrypted.
- In this project, it is necessary to retrieve information from a public API. In my opinion, a better approach would have been to generate a table in a database, migrate all the necessary information, and update it once or twice a month. This approach, not only helps to avoid getting many requests errors when accessing the public API or dealing with external issues but also, we improve the system performance.
- The application was designed with a focus on extensibility. Considerations were made to facilitate future adjustments. For instance, if there arises a need to manage not only USD clearing costs but also support other currencies in the future, an easy modification can be made within the system.
- I assumed that the CRUD for managing card-issuing countries includes validations, such as ensuring that the country can only be an ISO2 code or the word 'OTHER'.

Technical tools

- Spring boot 3.2.2
- Maven
- Java 17
- Postgres

Utilised libraries

Spring Web: Build web, including RESTful. Uses Apache Tomcat as the default embedded container.

Spring Data JPA: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

PostgreSQL Driver: A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standar, database independent Java code.

Spring Boot DevTools: Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Lombok : Java annotation library which helps to reduce boilerplate code.

Technical Decision

- I have chosen to use a layered architecture since it helps to organise and manage different aspects of the system independently. It makes it easier to update, maintain, and scale specific components without affecting the entire application.
- I chose Spring Boot as my framework because it comes with many helpful features. These features make it easier to concentrate on the business side of things and speed up the development process.
- I chose to make a REST API to make it highly available because it can grow horizontally.
- I tried to follow best practices as much as possible, such as practices of clean code, SOLID principles, ensuring functions and classes have a single responsibility, and following the Dependency Inversion Principle by promoting higher-level modules depending on abstractions (interfaces).

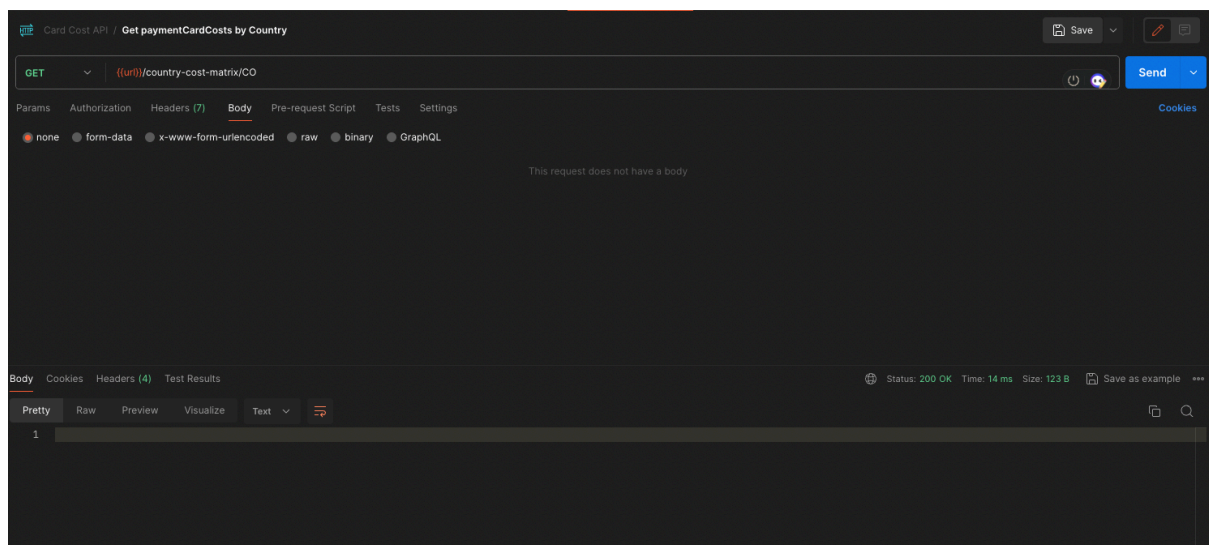


- I chose a relational database because, if we want to add more features in the future, such as incorporating costs for different card types (Visa, Master), or supporting additional types of currencies, it is more straightforward to achieve with relational databases in terms of consistency.
- I have decided to use the Repository pattern to simplify testing and centralise all data access operations, making it easier to manage, update, and troubleshoot database related issues.
- I decided to use Mockito and H2 as the in memory database for testing, simply because they make testing easier.
- I chose to handle the API calls in an external service using an interface (IApiCallService) to make the application more flexible. So, if we need to change the provider in the future, it can be done more easily without impacting the rest of the system.
- I opted to use a Global Exception Handler to handle try-catch blocks more easily and without cluttering the code.
- I tried to follow the best practices of REST API trying to return standar error codes, using nouns instead of verbs in endpoint paths.

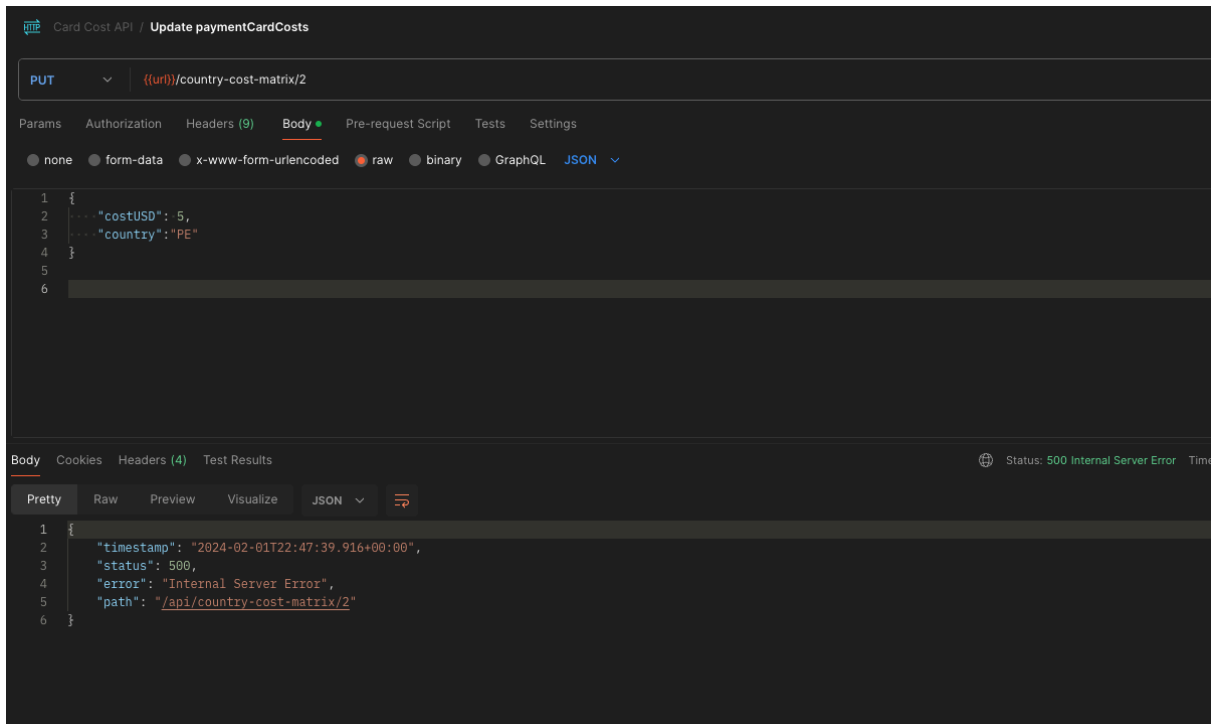
Areas for improvement

Due to the limited time I had for this test, there are areas that could be improved.

- I would have cached the IIN information in a Redis database to avoid making API calls if the migration approach is not chosen.
- I would have generated environments, such as `application.dev.properties` and `application.live.properties` in order to have the setting separately without having to change it manually.
- I would have the sensitive information outside the repository, project settings or docker file (using Docker secrets).
- If I had more time, I would like to add improvements such as versioning the API and implementing health checks, improve logs.
- I tested the application a lot to demonstrate different things that Spring Boot can do with testing. I tested calling APIs using a mock of RestTemplate and simulated the database with an H2 in-memory database. I tested all the layers, including controllers, repositories, and services, and also did integration tests for the Payment Card Cost Service. I wanted to show everything it can do, but if I had more time, I would have liked to try more unusual situations.
- Improve the response when retrieving the cost matrix by country. If the country doesn't exist, it should throw a 'not found' message instead of returning null and a 200 OK status.



- Make the message better when updating a country cost matrix. If a user tries to update with a country that already exists, get a more helpful message like a 500 internal error, instead of a 4xx error like Bad Request (400).



The screenshot shows a REST client interface for the 'Card Cost API' with the endpoint 'Update paymentCardCosts'. The request method is 'PUT' and the URL is '({url})/country-cost-matrix/2'. The request body is a JSON object:

```
{  "costUSD": 5,  "country": "PE"}
```

. The response status is '500 Internal Server Error'. The response body in JSON format is:

```
{  "timestamp": "2024-02-01T22:47:39.916+00:00",  "status": 500,  "error": "Internal Server Error",  "path": "/api/country-cost-matrix/2"}
```

- I would implement an authentication mechanism for the API, using keys, OAuth tokens, or certificates to ensure legitimate communication between services
- This improvement is not very crucial, but it could be a consideration. When the system receives the PAN, it is already checking the size using a regex expression. An additional improvement could be to verify if the PAN is correct using the Luhn algorithm. [\[2\]](#).

Instructions

Repository GitHub:

<https://github.com/vsseoane/Payment-Card-Cost-API>

In the project README, detailed instructions on how to run can be found.

Moreover, the Postman Collection to test the application can be found in the repo as a "Card Cost API.postman_collection" in the "Projects utils" folder . To set the Postman environment you can create a variable:

url = <http://localhost:8080/api> if you run the app in Docker or [url=http://localhost:8090/api](http://localhost:8090/api) if you run the application locally.

To run local

In application.properties:

```
server.port=8090
spring.datasource.url=jdbc:postgresql://localhost:5432/payments
spring.datasource.username=appuser
spring.datasource.password=admin
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show.sql=true
binTableApi.baseUrl=https://lookup.binlist.net/
```

To run with Docker

- To generate an image and a container:
 1. Generate package:
mvn clean package
 2. Build the app in Docker
docker compose build app
 3. Build and start services
docker compose up

- Use the Postman Collection to test it.

For more details, here there is a video which shows how it works.

<https://www.loom.com/share/448d6fc420cd410280bc1a59e058dc92?sid=1de76d24-af9c-41df-b9c4-66c0441e666d>