

Microservices Mini Project - 2

Hotel Room Booking Application

Project Overview

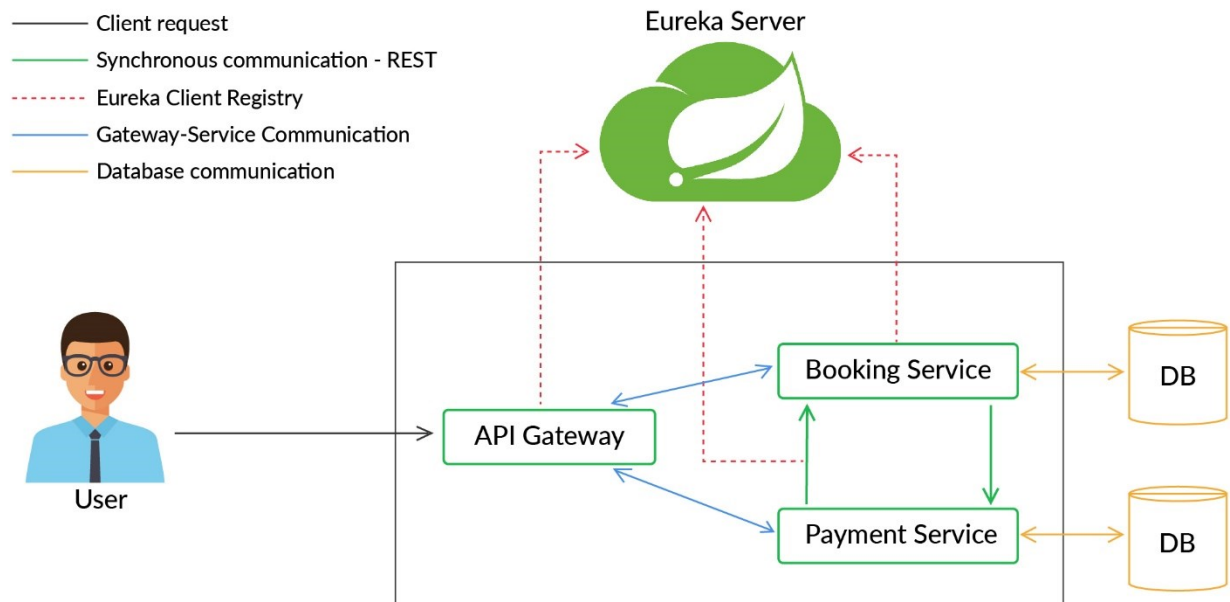
We are breaking the **Hotel room booking application** into three different microservices, which are as follows:

1. **API-Gateway** - This service is exposed to the outer world and is responsible for routing all requests to the microservices internally.
2. **Booking service** - This service is responsible for collecting all information related to user booking and sending a confirmation message once the booking is confirmed.
3. **Payment service** - This is a dummy payment service; this service is called by the booking service for initiating payment after confirming rooms.

Application Workflow

Let's now try to understand the workflow of the application using the following architecture diagram.

Note: The implementation details of the workflow will be discussed later in the problem statement.



- Initially, the API Gateway, Booking service and Payment service register themselves on the Eureka server. Note that the user does not directly interact with Booking or Payment service. All requests are sent to API Gateway which then sends the requests to the relevant microservice.
- The user initiates room booking using the 'Booking' service by providing information such as *toDate*, *fromDate*, *aadharNumber* and number of rooms required (*numOfRooms*).
- The 'Booking' service returns back the list of room numbers and price associated and prompts the user to enter the payment details if they wish to continue ahead with the booking. It also stores the details provided by the user in its database.
- If the user wants to go ahead with the booking, then they can simply provide the payment related details like *bookingMode*, *upild/cardNumber* to the 'Booking' service which will be further sent to the 'Payment' service. Based on this data, the payment service will perform a dummy transaction and return back the transaction Id associated with the transaction to the Booking service. All the information related to transactions is saved in the database of the payment service.
- Once the transaction completes, booking is confirmed and therefore Booking service sends a confirmation message on the console.

The rationale behind choosing synchronous communication (REST) between the Booking and Payment services is that the response('transactionId') is required before confirming the booking and sending a message.

Note: A schema for both Booking service & Payment service is attached along with this document.

1. Booking Service

This service is responsible for taking input from users like- *toDate*, *fromDate*, *aadharNumber* and the number of rooms required (*numOfRooms*) and save it in its database. This service also generates a random list of room numbers depending on 'numOfRooms' requested by the user and returns the room number list (*roomNumbers*) and total *roomPrice* to the user. The logic to calculate room price is as follows:

$$\text{roomPrice} = 1000 * \text{numOfRooms} * (\text{number of days})$$

Here, 1000 INR is the base price/day/room.

If the user wishes to go ahead with the booking, they can provide the payment related details like *bookingMode*, *upild / cardNumber*, which will be further sent to the payment service to retrieve the *transactionId*. This *transactionId* then gets updated in the Booking table created in the database of the Booking Service and a confirmation message is printed on the console.

1.1 Model Classes:

Refer to the "booking" table in the schema to create the entity class named "BookingInfoEntity".

1.2 Controller Layer:

Endpoint 1: This endpoint is responsible for collecting information like fromDate, toDate, aadharNumber, numOfRooms from the user and save it in its database.

- URI: /booking
- HTTP METHOD: POST
- RequestBody: fromDate, toDate, aadharNumber, numOfRooms
- Response Status: Created
- Response: ResponseEntity<BookingInfoEntity>

The screenshot displays a REST client interface. The top section shows a POST request to the URL `http://localhost:9191/hotel/booking...`. The 'Body' tab is selected, showing a JSON request body with the following fields: `fromDate` (2021-06-20), `toDate` (2021-06-25), `aadharNumber` (Sample-Aadhar-Number), and `numOfRooms` (3). The bottom section shows the response body in 'Pretty' format, which is a JSON object containing: `id` (1), `fromDate` (ISO timestamp), `toDate` (ISO timestamp), `aadharNumber` (Sample-Aadhar-Number), `roomNumbers` (68, 54, 91), `roomPrice` (15000), `transactionId` (0), and `bookedOn` (ISO timestamp). The status bar indicates a 201 Created response with a time of 612 ms and a size of 354 B.

```
POST http://localhost:9191/hotel/booking...

{
  "fromDate": "2021-06-20",
  "toDate": "2021-06-25",
  "aadharNumber": "Sample-Aadhar-Number",
  "numOfRooms": 3
}

{
  "id": 1,
  "fromDate": "2021-06-20T00:00:00.000+00:00",
  "toDate": "2021-06-25T00:00:00.000+00:00",
  "aadharNumber": "Sample-Aadhar-Number",
  "roomNumbers": "68,54,91",
  "roomPrice": 15000,
  "transactionId": 0,
  "bookedOn": "2021-06-02T13:55:54.885+00:00"
}
```

Note 1: The value of the transactionId returned is 0. It means that no transaction is made for this booking. Once the transaction is done, the transactionId field in the booking table will get replaced with the transactionId received from the Payment service.

Note 2: The room numbers displayed are not based on the availability of vacant rooms. They are rather randomly generated integers between 1 and 100. This is done to trim down the complexity of the problem statement.

Note 3: The field 'id' in the response body represents the 'BookingId'.

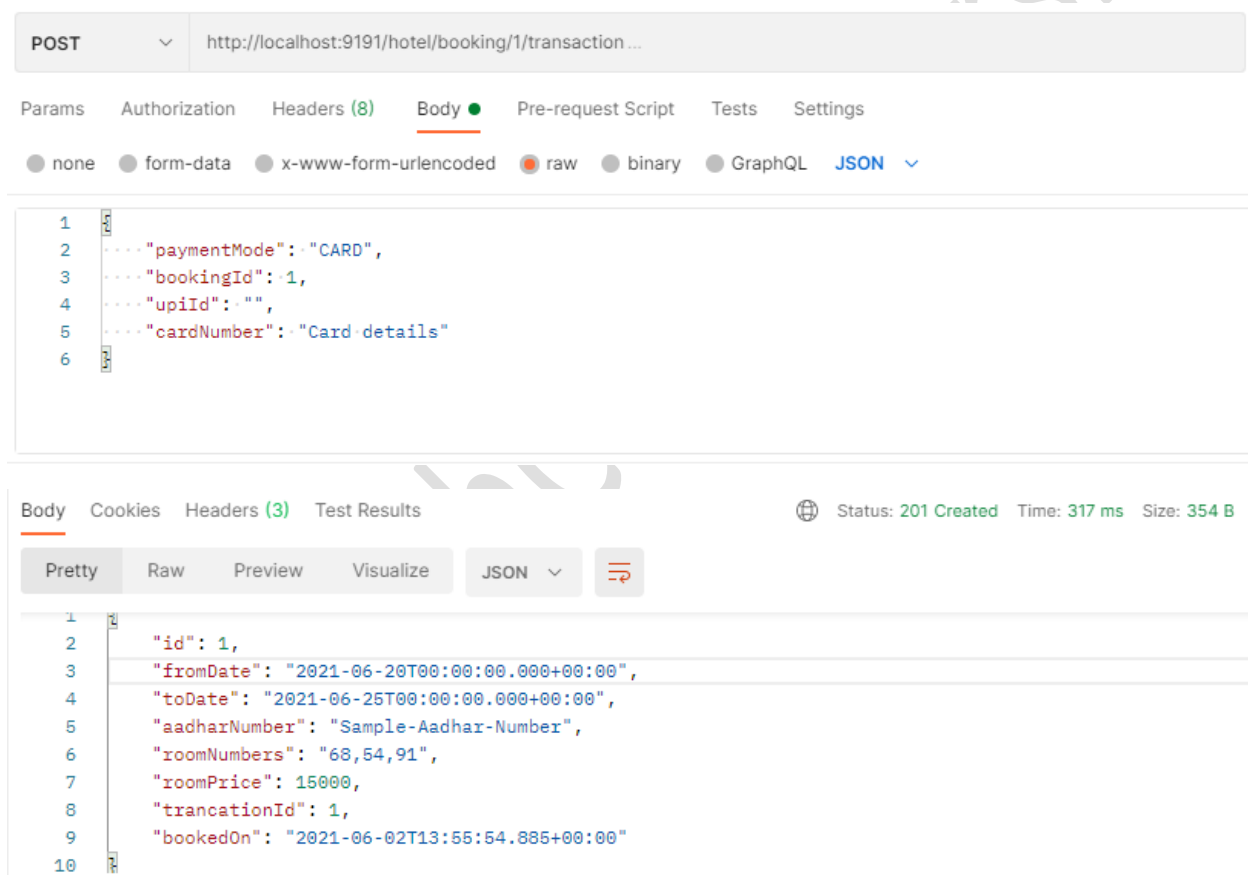
Endpoint 2: This endpoint is responsible for taking the payment related details from the user and sending it to the payment service. It gets the transactionId from the Payment service in response and saves it in the booking table. Please note that for the field 'paymentMode', if the user provides any input other than 'UPI' or 'CARD', then it means that the user is not interested in the booking and wants to opt-out.

URL: booking/{bookingId}/transaction

HTTP METHOD: POST

PathVariable: int

RequestBody: paymentMode, bookingId, upid, cardNumber



The screenshot displays a REST client interface. The top section shows a POST request to the URL `http://localhost:9191/hotel/booking/1/transaction...`. The 'Body' tab is selected, showing a JSON request with the following content:

```
1 {
2   "paymentMode": "CARD",
3   "bookingId": 1,
4   "upiId": "",
5   "cardNumber": "Card details"
6 }
```

The bottom section shows the response body, which is a JSON object with the following content:

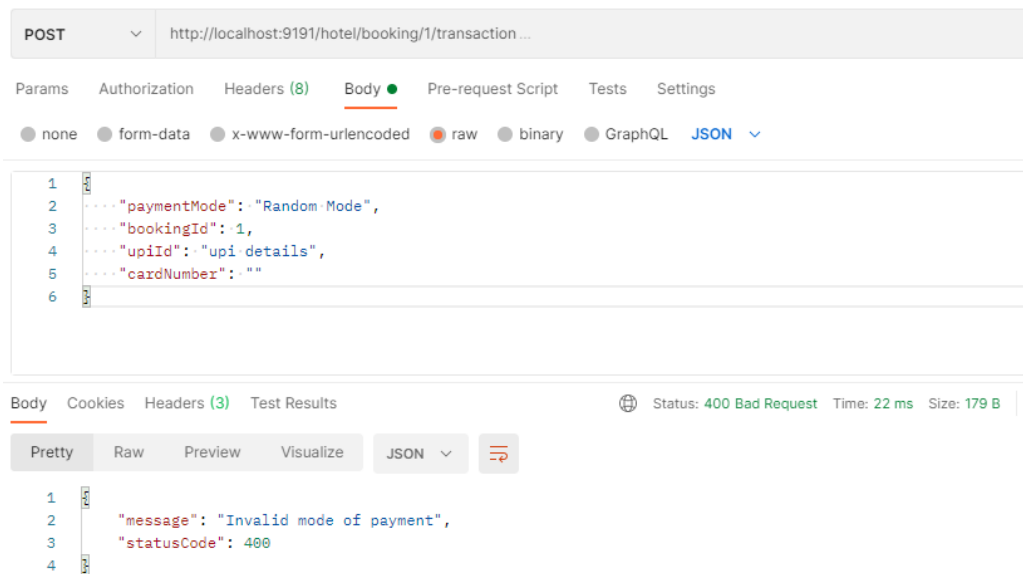
```
1 {
2   "id": 1,
3   "fromDate": "2021-06-20T00:00:00.000+00:00",
4   "toDate": "2021-06-25T00:00:00.000+00:00",
5   "aadharNumber": "Sample-Aadhar-Number",
6   "roomNumbers": "68,54,91",
7   "roomPrice": 15000,
8   "transactionId": 1,
9   "bookedOn": "2021-06-02T13:55:54.885+00:00"
10 }
```

The response status is 201 Created, with a time of 317 ms and a size of 354 B.

Note that the transaction Id this time stores the actual transactionId associated with the transaction.

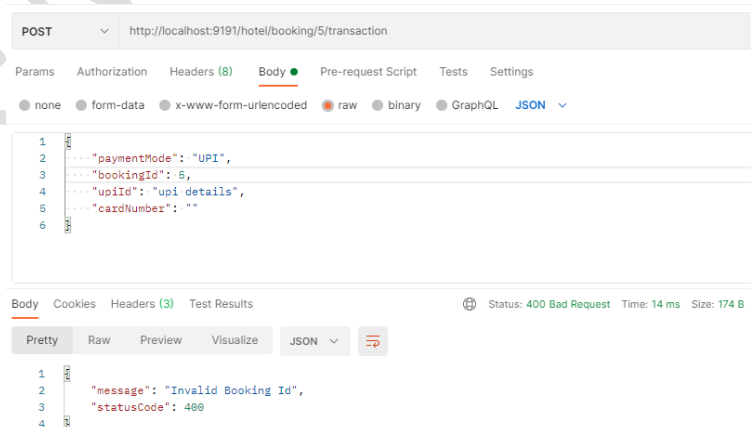
Exception 1: If the user gives any other input apart from “UPI” or “CARD”, the response message should look like the following:

```
1. {  
2.   "message": "Invalid mode of payment",  
3.   "statusCode": 400  
4. }  
5.
```



Exception 2: If no transactions exist for the Booking Id passed to this endpoint then the response message should look like the following:

```
1. {  
2.   "message": " Invalid Booking Id ",  
3.   "statusCode": 400  
4. }  
5.
```



1.3 Configure this service to run on port number 8081.

1.4 Configure the hotel booking service as Eureka Client

Once the configuration is done properly for this service, run the Eureka server, API Gateway and Booking service on your localhost.

1. Payment Service:

This service is responsible for taking payment-related information- paymentMode, upild or cardNumber, bookingId and returns a unique transactionId to the booking service. It saves the data in its database and returns the transactionId as a response.

1.1 Model Classes:

Refer to the "transaction" table in the schema to create the entity class named "TransactionDetailsEntity".

1.2 Controller Layer:

Endpoint 1: This endpoint is used to imitate performing a transaction for a particular booking. It takes details such as bookingId, paymentMode, upild or cardNumber and returns the transactionId automatically generated while storing the details in the 'transaction' table. Note that this 'transactionId' is the primary key of the record that is being stored in the 'transaction' table.

After receiving the transactionId from 'Payment' service, confirmation message is printed on the console.

Message String:

```
String message = "Booking confirmed for user with aadhaar number: "
+ bookingInfo.getAadhaarNumber()
+ " | "
+ "Here are the booking details: " + bookingInfo.toString();
```

Note: This endpoint will be called by the 'endpoint 2' of the Booking service.

- URL: /transaction
- HTTP METHOD: POST
- RequestBody: paymentMode, bookingId, upild, cardNumber

POST http://localhost:9191/payment/transaction

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1
2  .... "paymentMode": "CARD",
3  .... "bookingId": 1,
4  .... "upiId": "",
5  .... "cardNumber": "Card details"
6
```

- Response Status: Created
- Response: ResponseEntity<transactionId>

Body Cookies Headers (3) Test Results Status: 201 Created Time: 35 ms Size: 122 B

Pretty Raw Preview Visualize JSON

```
1 2
```

EndPoint 2: This endpoint presents the transaction details to the user based on the transactionId provided by the user.

- URL: /transaction/{transactionId}
- HTTP METHOD: GET
- RequestBody: (PathVariable) int
- Response Status: OK
- Response: ResponseEntity<TransactionDetailsEntity>

GET http://localhost:9191/payment/transaction/1

Params Authorization Headers (6) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (3) Test Results Status: 200 OK Time: 9 ms Size: 198 B

Pretty Raw Preview Visualize JSON

```
1
2  "id": 1,
3  "paymentMode": "CARD",
4  "bookingId": 1,
5  "upiId": "",
6  "cardNumber": "Card details"
7
```

1.3 Configure the service to run on port 8083.

1.4 Configure the service as a Eureka client.

Once the Eureka client configuration is done properly for this service,run the Eureka server, API Gateway, Booking service and Payment service on your localhost.

Following are the dependencies for each service:

- **Booking Service:**
 - Spring Cloud Netflix Eureka Client
 - Spring Boot Web
 - Spring Boot Data JPA
 - Spring Boot Devtools
- **Payment Service:**
 - Spring Cloud Netflix Eureka Client
 - Spring Boot Web
 - Spring Boot Data JPA
 - Spring Boot Devtools
- **API Gateway:**
 - Dependencies: Spring Boot Actuator, Spring Cloud Netflix Eureka Client
 - Spring Boot Devtools
 - Configure properties
- **Eureka Server:**
 - Dependencies: Spring Cloud Netflix Eureka Client, Eureka Discovery Server
 - Spring Boot Devtools
 - Open the Eureka server and annotate the main class with proper annotation so that the Eureka server gets enabled.
 - Set properties for running standalone Eureka servers.
 - Set port for Eureka server as 8761.

Please note that the dependencies list mentioned above is not exhaustive in nature. Depending upon your logic and implementation, several additional dependencies might get added. These dependencies are shared just to help you remember the most fundamental ones used in the project.

Once the configuration is done properly for this service, run the Eureka server and API Gateway service on your local host.

Deliverables:

1. A github link (private repository) containing a folder named “Sweet-home” containing all the codes, i.e. project of each service- booking, payment, API Gateway(optional) and Eureka server. Keep all the API endpoint screenshots in a readme file or in the PDF document.
2. A PDF document (CodeLogic.pdf) containing the brief walkthrough of the logic that you applied to solve the entire project. You can specify this service wise. This will also have the required instruction for the graders to run the entire project on any particulate machine. Note: There is no specific format for this document.

Evaluation Rubrics		
Sr. No.	Criteria	Expectation
1.	Code Readability and Guidelines(5%)	<ul style="list-style-type: none">• The code is formatted correctly. It uses logical spacing and indentations.• The code contains useful comments, which explain how the complicated portions of the code work.• Functions and variables have proper and logical names.
2.	Register services on Eureka Server(10%)	<ul style="list-style-type: none">• The Booking and payment service are properly registered on Eureka and are reflected in the Eureka UI.
3.	Are all the APIs of the Booking service working as expected? (50%)	<ul style="list-style-type: none">• Code can fetch the user data related to booking properly.• Code can generate the room numbers allocated to the user based on the number of rooms(numOfRooms) required and save it in the database.• The ‘bookedOn’ date is the present date.• The roomPrice is calculated properly based on the fromDate and toDate.

		<ul style="list-style-type: none"> • Code can fetch the payment-related information from the user while handling the exception related to paymentMode properly, as mentioned in the problem statement. • Exception related to bookingId as mentioned is handled properly. • REST Template is used to establish synchronous communication between the hotel booking and payment services. • TransactionId is updated in the booking table. • The confirmation message is properly displayed in the format mentioned in the problem statement. • Model class corresponds to the provided schema.
4.	<p>Are all the APIs of the Payment service working as expected?</p> <p>(35%)</p>	<ul style="list-style-type: none"> • Code can retrieve the payment-related information obtained from the booking service and save it in its own database properly. • The 'bookingId' in the database is correctly mapped to the obtained 'bookingId' from the booking service. • Code can return the transaction details to the user based on the transactionId provided by the user. • Model class corresponds to the provided schema.

HAPPY CODING