Tasks

In this segment, the complete description of the project is specified. Before you proceed with

the tasks, please take a look at the schema of the database for each service.

Sweet-Home Schema

Download

1. Create a folder "Sweet-Home".

NOTE: Please use the same naming convention as specified in the problem statement and schema.

2. Create **project structure** using Spring Initializer for the microservices- 'Booking' and 'Payment' - and 'Eureka' server inside "Sweet-Home".

Spring Initializer URL: https://start.spring.io/

Note that the 'Notification' service will not be a Spring project, rather it will be a Maven-based Java project which will simply subscribe to Kafka topic.

Following are the dependencies for each service:

a. Booking Service:

- Eureka Discovery Client
- Spring Data JPA
- MySQL Driver
- Spring Web
- Add the following dependency in pom.xml.

```
<dependency>
            <groupId>org.apache.kafka</groupId>
                    <artifactId>kafka-clients</artifactId>
            <version>2.7.0</version>
 </dependency>
```

b. Payment Service:

- Eureka Discovery Client
- Spring Data JPA
- MySQL Driver
- Spring Web

c. Notification Service:

It is a simple **Java-Maven project** that subscribes to a particular Kafka topic. You can download the 'pom.xml' file for this service from the bottom of this page. Please note that the pom.xml is configured for java version 11 and if you are working on a different version, please modify the file accordingly.

d. Eureka Server:

- Dependencies: Eureka Server
- 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.

3. Creation of RDS instance:

Create a single RDS instance and create separate databases to host the booking and transaction table as per the **Sweet-Home** schema. Please make sure that the MySQL database

on AWS RDS is set up before starting. You can refer to the documents given below for the set-up.

RDS Set Up

Download

Important Note:

Amazon RDS is a costly service of AWS. Hence to avoid burning up your monthly AWS budget, Please make sure to Delete your RDS DB Instance once done. If you are not planning to use your RDS DB instance (After the project or you have 2-3 days break) for a while we highly recommend you delete the database in that case. Please follow the below steps to delete the RDS DB instance. This must be done to avoid burning your AWS budgets.

RDS Delete

Download

4. 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 RDS 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 formulae to calculate room price is as follows:

roomPrice = **1000*** numOfRooms*(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 paymentMode, upiId / 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 RDS database of the Booking service and a notification is sent to the user about their booking confirmation.

A sample code that you could refer to create random room numbers is as follows:

/* The following code snippet returns a random number list with upperbound of 100 and 'count' number of entries in the number list*/

```
public static ArrayList<String> getRandomNumbers(int count){
        Random rand = new Random();
        int upperBound = 100;
        ArrayList<String>numberList = new ArrayList<String>();

       for (int i=0; i<count; i++){
                   numberList.add(String.valueOf(rand.nextInt(upperBound)));
        }

         return numberList;
}
```

Output:

```
for count=5, numberList contains any 5 numbers between 0 and 100
```

4.1 Model Classes:

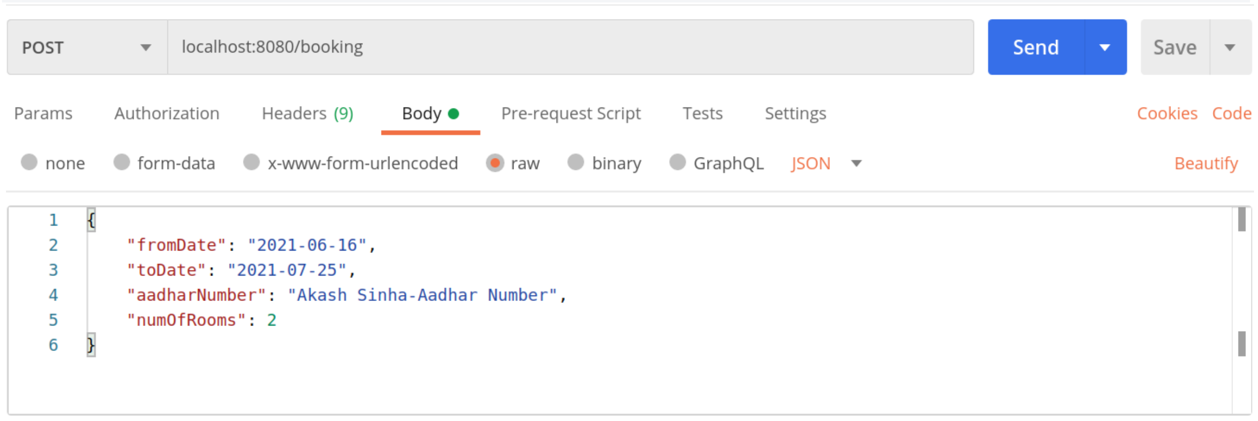
Refer to the "booking" table in the schema to create the entity class named

"**BookingInfoEntity**".

4.2 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.

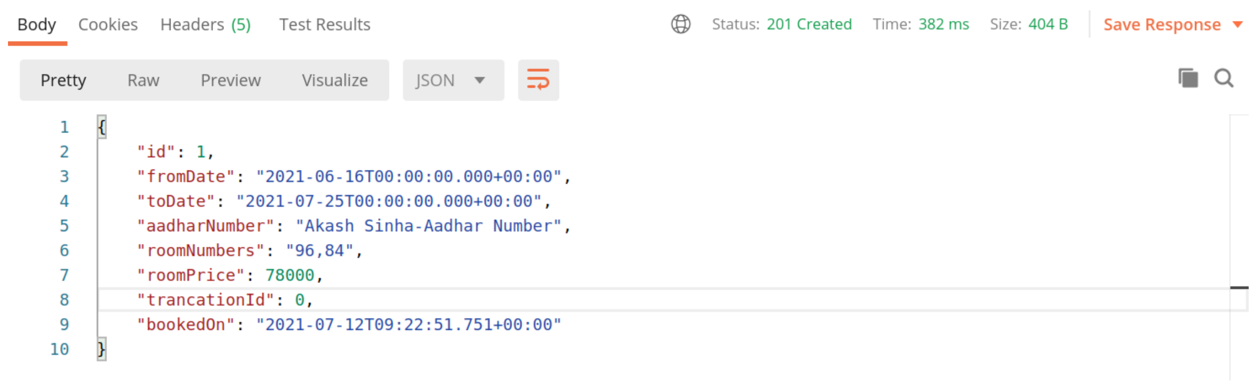
- **URI**: /booking
- **HTTP METHOD**: POST
- **RequestBody**: fromDate, toDate,aadharNumber,numOfRooms



```json
{
    "fromDate": "2021-06-16",
    "toDate": "2021-07-25",
    "aadharNumber": "Akash Sinha-Aadhar Number",
    "numOfRooms": 2
}
```

Figure 1: Request Body

- **Response Status:** Created
- **Response**: ResponseEntity<BookingInfoEntity>


Body Cookies Headers (5) Test Results Status: 201 Created Time: 382 ms Size: 404 B Save Response

Pretty Raw Preview Visualize JSON

```
{
    "id": 1,
    "fromDate": "2021-06-16T00:00:00.000+00:00",
    "toDate": "2021-07-25T00:00:00.000+00:00",
    "aadharNumber": "Akash Sinha-Aadhar Number",
    "roomNumbers": "96,84",
    "roomPrice": 78000,
    "trancationId": 0,
    "bookedOn": "2021-07-12T09:22:51.751+00:00"
}
```

Figure 2: Response

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.

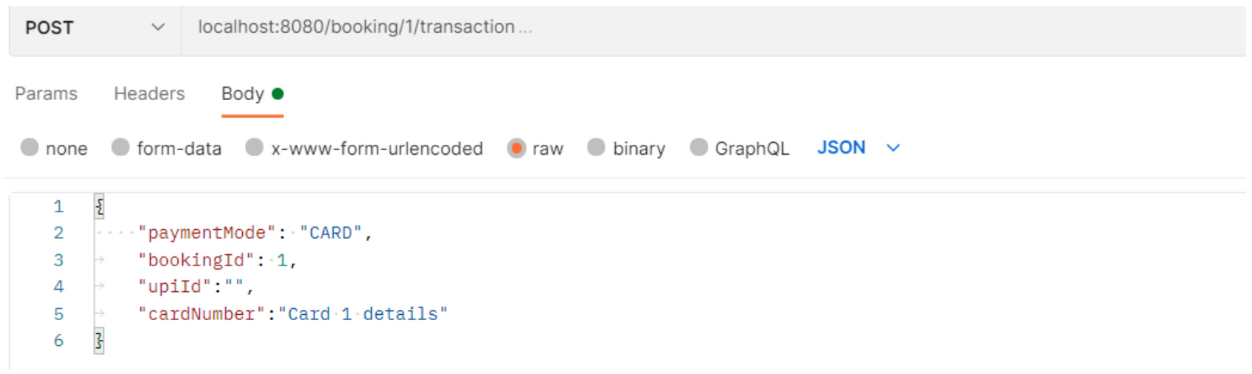
Note 2: The room numbers displayed are not based on the availability of vacant rooms. They are rather randomly generated integers between 1 and 100. This is done to trim down the complexity of the problem statement.

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.

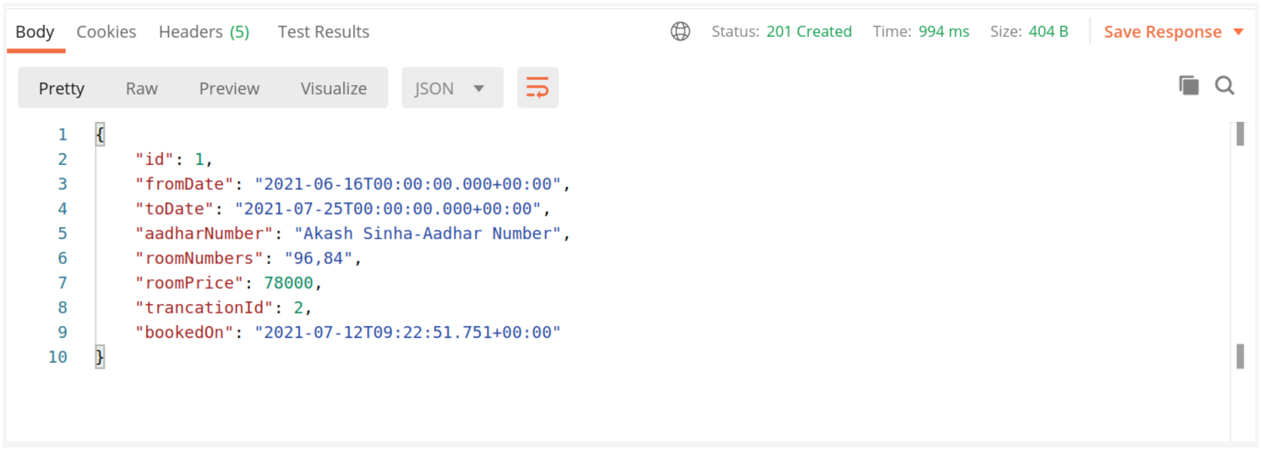
- **URL**: booking/{bookingId}/transaction
- **HTTP METHOD**: POST
- **PathVaraiable**: int
- **RequestBody**: paymentMode, bookingId,upiId,cardNumber



POST localhost:8080/booking/1/transaction ...

```
{
    "paymentMode": "CARD",
    "bookingId": 1,
    "upiId":"",
    "cardNumber":"Card 1 details"
}
```

Figure 3: Request Body



Status: 201 Created Time: 994 ms Size: 404 B

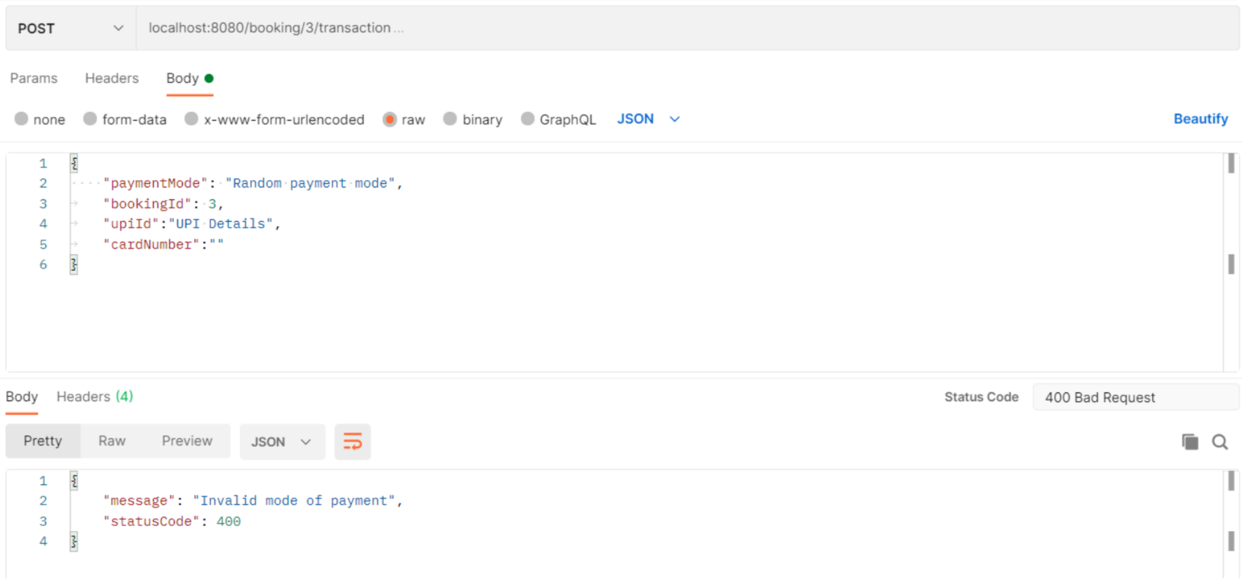
```
{
    "id": 1,
    "fromDate": "2021-06-16T00:00:00.000+00:00",
    "toDate": "2021-07-25T00:00:00.000+00:00",
    "aadharNumber": "Akash Sinha-Aadhar Number",
    "roomNumbers": "96,84",
    "roomPrice": 78000,
    "trancationId": 2,
    "bookedOn": "2021-07-12T09:22:51.751+00:00"
}
```

Figure 4: Response

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:

```
{
  "message": "Invalid mode of payment",
 "statusCode": 400
}
```



POST localhost:8080/booking/3/transaction ...

```
{
    "paymentMode": "Random payment mode",
    "bookingId": 3,
    "upiId":"UPI Details",
    "cardNumber":""
}
```

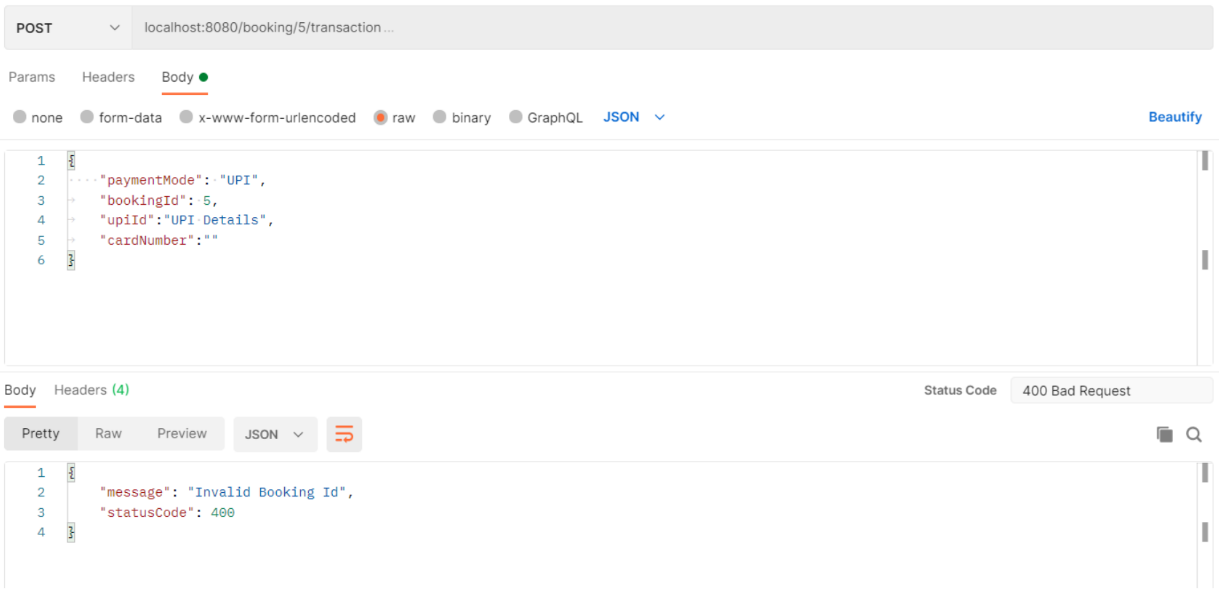
Status Code 400 Bad Request

```
{
    "message": "Invalid mode of payment",
    "statusCode": 400
}
```

Figure 5: Exception 1

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

```
{
  "message": " Invalid Booking Id ",
 "statusCode": 400
}
```



POST localhost:8080/booking/5/transaction

```
{
    "paymentMode": "UPI",
    "bookingId": 5,
    "upiId":"UPI Details",
    "cardNumber":""
}
```

Status Code 400 Bad Request

```
{
    "message": "Invalid Booking Id",
    "statusCode": 400
}
```

Figure 6: Exception2

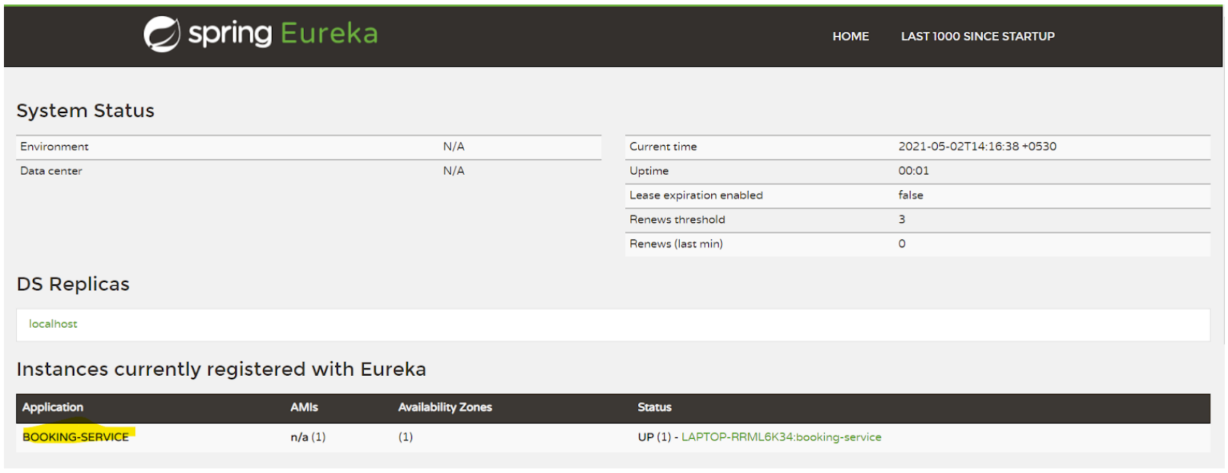
4.3 Configure this service to run on port number 8080.

4.4 Configure the hotel booking service as Eureka Client

Once the configuration is done properly for this service, run the Eureka server and Booking service on your localhost. Post this when you hit the Eureka server IP from your Internet Explorer, you will see something similar.

Address:

http://localhost:8761/



System Status

Environment	N/A
Data center	N/A

Current time	2021-05-02T14:16:38 +0530
Uptime	00:01
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BOOKING-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-RRML6K34:booking-service

Figure 7: Eureka UI

5. Payment Service:

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

5.1 Model Classes:

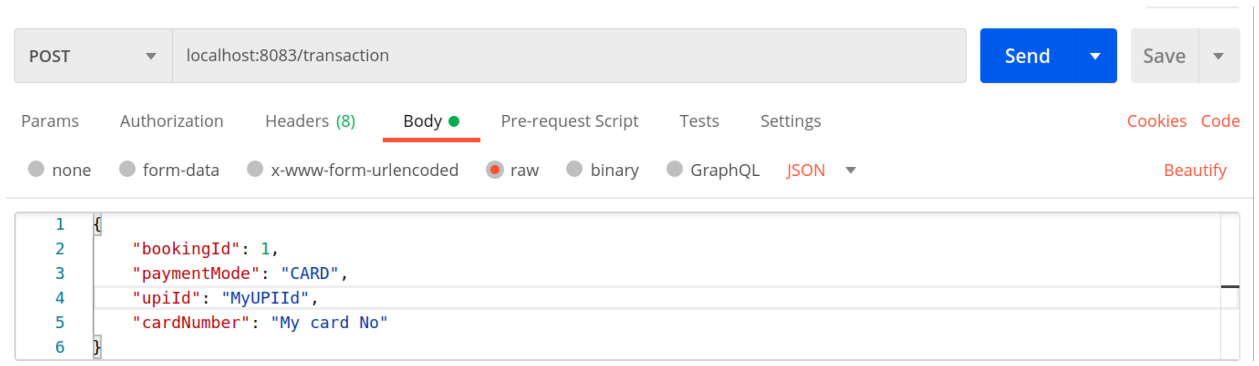
Refer to the “transaction” table in the schema to create the entity class named “TransactionDetailsEntity”.

5.2 Controller Layer:

Endpoint 1: This endpoint is used to imitate performing a transaction for a particular booking. It takes details such as bookingId, paymentMode,upiId 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.

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

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



```
{
    "bookingId": 1,
    "paymentMode": "CARD",
    "upiId": "MyUPIId",
    "cardNumber": "My card No"
}
```

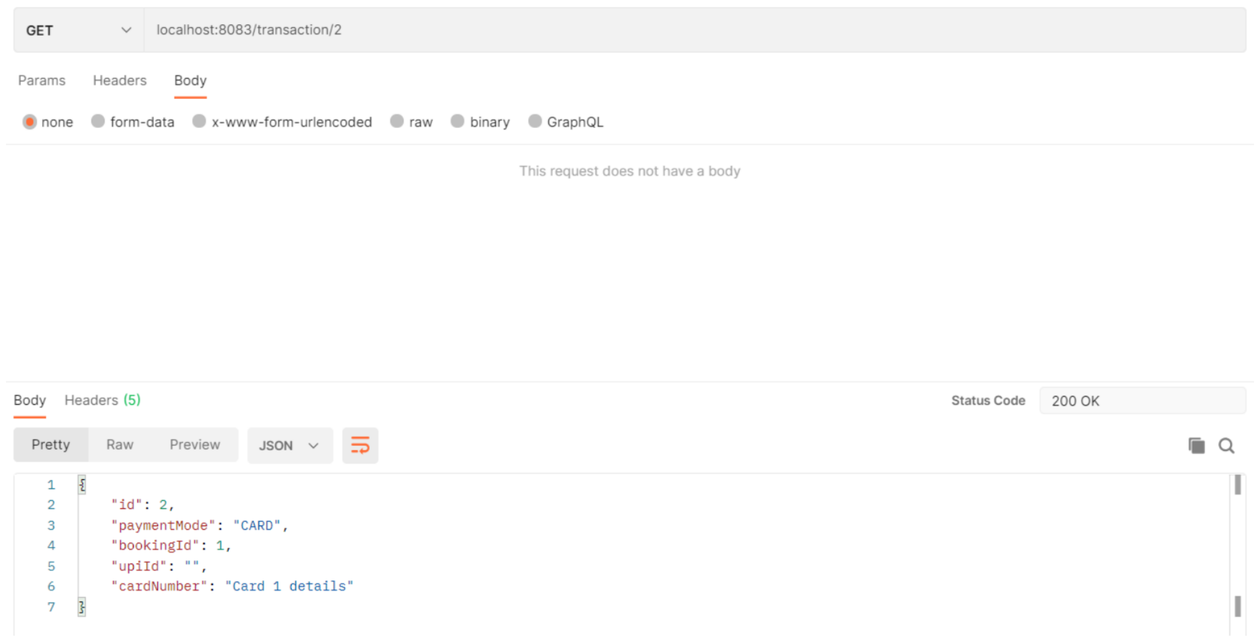
Figure 8: Request Body

Response Status: Created

Response: ResponseEntity<transactionId>

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**: (PathVaraiable) int
- **Response Status**: OK
- **Response:** ResponseEntity<TransactionDetailsEntity>


GET localhost:8083/transaction/2

Body: none

This request does not have a body

Status Code 200 OK

```
{
    "id": 2,
    "paymentMode": "CARD",
    "bookingId": 1,
    "upiId": "",
    "cardNumber": "Card 1 details"
}
```

Figure 9: Response

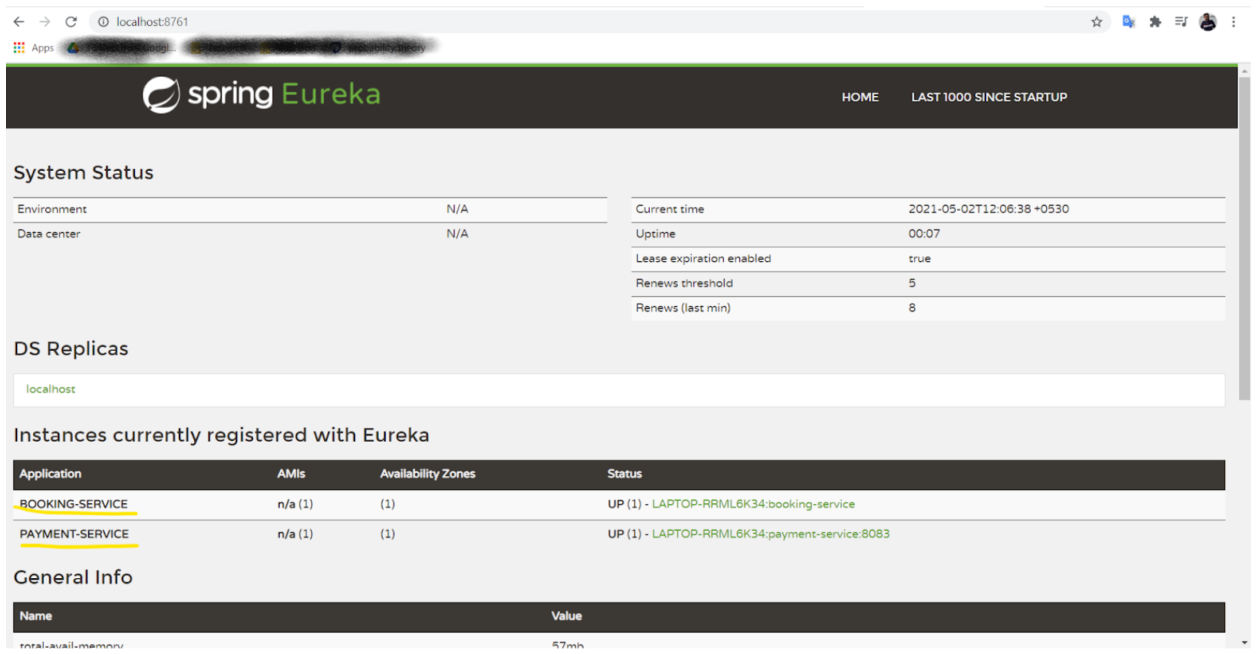
5.3 Configure the service to run on port 8083.

5.4 Configure the service as a Eureka client.

Once the Eureka client configuration is done properly for this service, and the Eureka server and Booking service is running on your localhost, you will see something similar when you hit the Eureka server IP from your Internet Explorer

Address:

http://localhost:8761/


localhost:8761

Apps [redacted]

spring Eureka

HOME LAST 1000 SINCE STARTUP

System Status

Environment	N/A
Data center	N/A

Current time	2021-05-02T12:06:38 +0530
Uptime	00:07
Lease expiration enabled	true
Renews threshold	5
Renews (last min)	8

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BOOKING-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-RRML6K34:booking-service
PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-RRML6K34:payment-service:8083

General Info

Name	Value
total-avail-memory	57mb

Figure 10: Eureka

Hint: Synchronous communication between booking and payment service has to be set up using the REST template of Spring Boot. Please refer to the module of " Discovery and communication between Microservices" for the same.

6. Notification Service:

This service consumes the messages published by the Booking service on Kafka and prints the same on the console. This service will be created while establishing asynchronous communication using Kafka.

6.1 Kafka Pre-requirement:Run Kafka on an AWS EC2 instance.

- Please navigate to Segment 7 of Session 2 of module '**Asynchronous communication using messaging models**' to get the detailed steps to run Kafka on EC2. Also, you can refer to the following documents to set up IP on EC2 and for starting the zookeeper and Kafka server.

My IP setup on EC2

Download

Kafka Quickstart

Download

Create a topic named **'message'**.

Note: Please keep your EC2 instance in the stopped state once you are done for the day.

6.2 Configure hotel booking service as a publisher

- In the Config package of hotel booking service, create a class name KafkaConfig.java
- Annotate it with proper annotation for making it a Configuration class
- Create a Bean that sets up all properties required by the Kafka Client
- Return a producer <String, String> instance from this bean
- Autowire Producer in BookingService class
- After saving the transactionId in the booking table, publish a message to Kafka on the message topic
- **Message String:**

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

6.3 Configure Notification service as a subscriber:

- In the Notification Service project, create Consumer Class and the main method.
- Set all properties required by the Kafka consumer.
- Subscribe to the “message” topic.
- Start consuming messages in a forever loop for notification service using Kafka.
- Start printing record values on the console.

Note: Always use Constructor Autowiring only for Autowiring. The reason for this is that it does not let objects be created unless the dependency exists. This enhances security.