Name: Yash Jayeshkumar Pandya
Email : yashp614929@gmail.com
Project : Course Project: Hotel Room Booking Application

Overview:

The project ‘Hotel Room Booking Application’ is used to book rooms for a single hotel. In this project, I have created a microservices based backend application using spring boot framework.

The project is divided into three different microservices, which are as follows:

Booking service

This service is exposed to the outer world and is responsible for collecting all information related to user booking.

Payment service

This is a dummy payment service; this service is called by the booking service for initiating payment after confirming rooms.

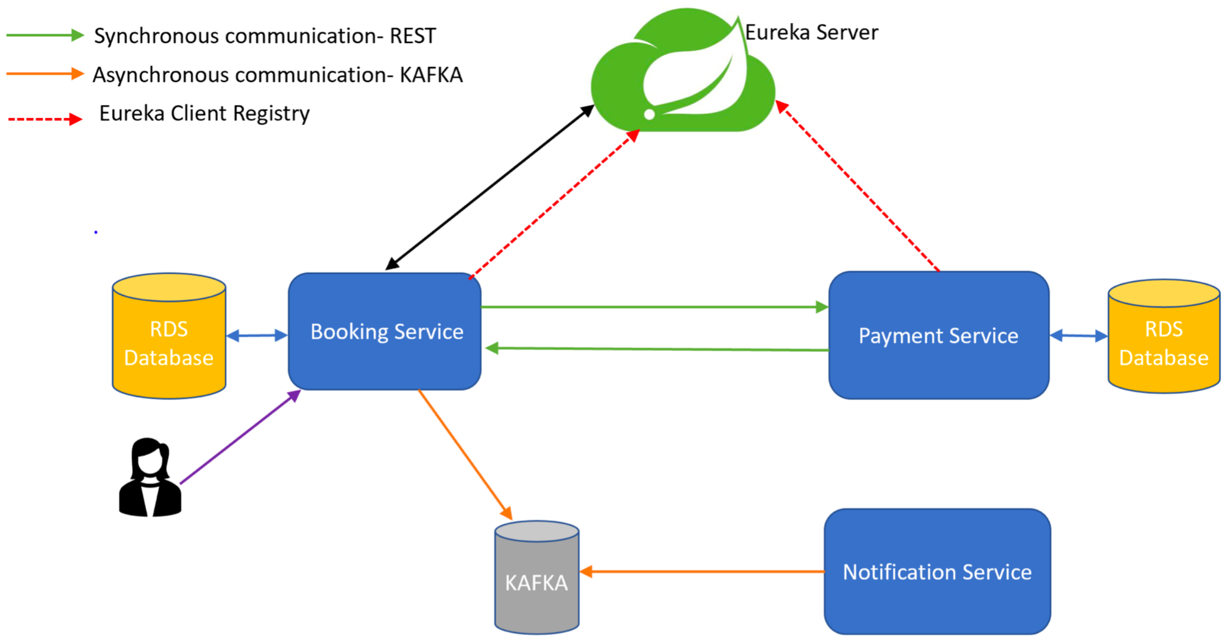
Notification service

When payment is done and all data is saved properly, the booking service puts a message object on the Kafka topic, and then, the notification service reads that message from the Kafka topic and prints the message to the console.

Application Workflow:

Initially, the Booking service and Payment service register themselves on the Eureka server. Note that the Booking service interacts with the Notification service asynchronously and therefore, the Notification service need not register itself as a Eureka client.

The user initiates room booking using the 'Booking' service by providing information such as toDate, fromDate, aadharNumber and number of rooms required (number of Rooms).



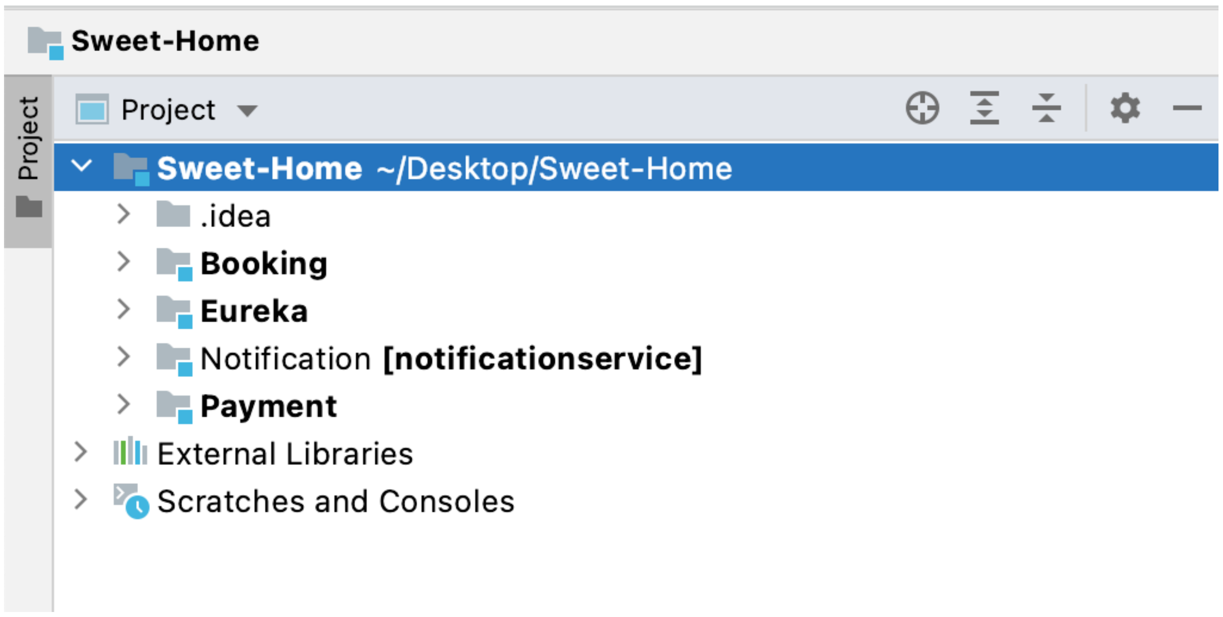
The 'Booking' service returns back the list of room numbers and price associated and prompts the user to enter the payment details if they wish to continue ahead with the booking. It also stores the details provided by the user in its database.

If the user wants to go ahead with the booking, then they can simply provide the payment related details like bookingMode, upiId/ 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 RDS database of the payment service.

With the help of Kafka and the Notification service, we send a notification to the user after the transaction is confirmed. (prints on the console of 'Notification' service, in this case)

Architecture:

All Services are combined into a single directory called ‘Sweet-Home’. Booking service, Payment service and Eureka are a type of Spring boot application whereas Notification service is a simple java maven project.


Sweet-Home

Project

Sweet-Home ~/Desktop/Sweet-Home
- .idea
- **Booking**
- **Eureka**
- Notification **[notificationservice]**
- **Payment**

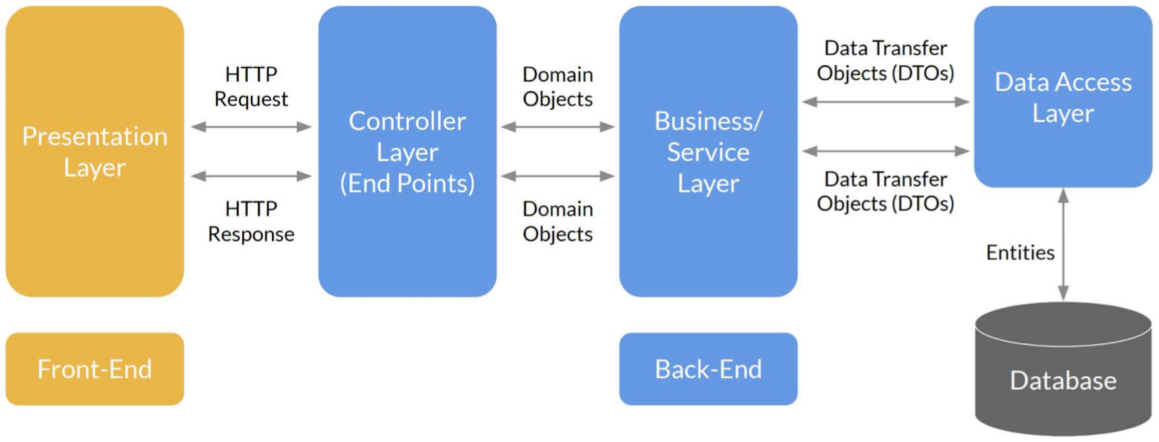
External Libraries

Scratches and Consoles

AWS RDS is used for database operations and services. To communicate with notification service from booking service, apache kafka is used on the AWS EC2 instance.

Booking and Payment Service internal architecture

- Based on Schema definition, Entity class is built
- Data Access Layer (DAO), a mediator between Database and Service Layer, through Data Transfer Objects (DTO), DAO is defined as a repository into the project.
- Service Layer acts as an interface between Controller and DAO.
- Controller layer interacts with presentation layer, through HTTP Requests



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 below code generates the random numbers in the booking service

```
// generate random numbers based on provided total number count
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;
}
```

Based on the schema definitions of the booking table, BookingInfoEntity class is created. The class is annotated with @Entity for the microservice to identify it as an entity and spring can create a table in the configured database for the same. The getter and setter methods along with toString are also created.

```
// booking table entity class to map to and from database
@Entity(name="booking")
@JsonPropertyOrder({"id"})
public class BookingInfoEntity {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @JsonProperty("id")
   private int bookingId ;

   @Column(nullable = true)
   private Date fromDate;

   @Column(nullable = true)
   private Date toDate;

   @Column(nullable = true)
   private String aadharNumber ;
   @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
   private int numOfRooms ;

   private String roomNumbers ;

   @Column(nullable = false)
   private int roomPrice ;

   private int transactionId=0;

   @Column(nullable = true)
   private Date bookedOn;

}
```

Repository is an interface which extends JPA Repository, which persists Java Objects into relational databases.

```
@Repository
public interface BookingRepository extends JpaRepository<BookingInfoEntity,
Integer> {
}
```

DTO class is independently worked as an interface among Repository and service layers. This is to keep the Database some portion of the back end isolated and unexposed.

BookingDTO class is used to map the request/response for the */bookin*g POST API.

```
// DTO class which is used to map Booking detail
@JsonPropertyOrder({"id"})
public class BookingDTO {

   @JsonProperty("id")
   private int bookingId ;

   private Date fromDate;

   private Date toDate;

   private String aadharNumber ;

   @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
   private int numOfRooms ;

   private String roomNumbers ;
```

```
    private int roomPrice ;

    private int transactionId=0;

    private Date bookedOn;

    @JsonProperty("id")
    public int getBookingId() { return bookingId; }

}
```

The another DTO class named BookingTransactionDTO class is used to map for the */booking/{bookingId}/transaction* POST API

```
// DTO class which is used to map booking transaction
public class BookingTransactionDTO {

    @NotNull
    private String paymentMode;

    @NotNull
    private int bookingId;

    private int transactionId;

    private String upiId ;

    private String cardNumber ;

    @JsonProperty("id")
    public void setTransactionId(int transactionId) {
        this.transactionId = transactionId;
    }
}
```

The ErrorDTO class is used to map the error/validation response.

```
// DTO class which is used to map exception/error
public class ErrorDTO {

    private String message;

    private int statusCode ;

}
```

On the other hand, KafkaNotificationDTO class is used to map the request after booking successfully completed by the kafka producer which is written in config/KafkaConfig class.

```java
// DTO class which is used to map kafka notifications
public class KafkaNotificationDTO {

   private String topic ;

   private String key ;

   private String value ;
}
```

Config directory inside the Booking service contains the KafkaInfoProducer interface which defines the gerProducer method of apache kafka, following the implementation is written in the KafkaConfig class which overrides the getProducer method and returns the kafka producer instance to BookingService class for further use.

```java
public interface KafkaInfoProducer {

   Producer<String, String> getProducer() throws IOException;

}

// Kafka Implementation
@Component
public class KafkaConfig implements KafkaInfoProducer {

   @Override
   public Producer<String, String> getProducer() throws IOException {
       Properties properties = new Properties();
       properties.put("bootstrap.servers",
            "ec2-174-129-48-56.compute-1.amazonaws.com:9092");
       properties.put("acks", "all");
       properties.put("retries", 0);
       properties.put("linger.ms", 0);
       properties.put("partitioner.class",
            "org.apache.kafka.clients.producer.internals.DefaultPartitioner");
       properties.put("key.serializer",
            "org.apache.kafka.common.serialization.StringSerializer");
       properties.put("value.serializer",
            "org.apache.kafka.common.serialization.StringSerializer");
       properties.put("request.timeout.ms", 30000);
       properties.put("timeout.ms", 30000);
       properties.put("max.in.flight.requests.per.connection", 5);
       properties.put("retry.backoff.ms", 5);
```

```
        //Instantiate Producer Object
         return new KafkaProducer<>(properties);

    }
}
```

Booking controller is the interface between the presentation layer and the service class. This class has the POST methods which take the request input and send it to the service class and reverse. This class is also responsible for sending the response back in the HTTP form. it converts DTO to Entity and Entity back to DTO and sends either response or error back with appropriate status code.

```
// POST API to create booking and store into Database
// It will return booking detail data as a response
@PostMapping(value="/booking", consumes = MediaType.APPLICATION_JSON_VALUE,
       produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity createBooking(@RequestBody BookingDTO bookingDTO){

   BookingInfoEntity booking = modelMapper.map(bookingDTO,
BookingInfoEntity.class);
   BookingInfoEntity savedBooking = bookingService.makeBooking(booking);
   BookingDTO savedbookingDTO = modelMapper.map(savedBooking,
BookingDTO.class);
   return new ResponseEntity(savedbookingDTO, HttpStatus.CREATED);
}



// POST API to confirm booking and store into Database as a booking
transaction
// It will return booking transaction detail data as a response
@PostMapping(value="/booking/{bookingId}/transaction", consumes =
MediaType.APPLICATION_JSON_VALUE,
       produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity confirmBooking(
       @PathVariable(name = "bookingId")  int bookingId,
       @RequestBody BookingTransactionDTO bookingTransactionDTO) throws
IOException {

   BookingInfoEntity savedBooking =
bookingService.updateBookingByCreatingTransaction(bookingId,
bookingTransactionDTO);
   String message = "Booking confirmed for user with aadhaar number: " +
savedBooking.getAadharNumber() +    "    |    "  + "Here are the booking
details:    " + savedBooking.toString();

   // kafka notification send after successful transaction
   bookingService.sendNotification("message", "notification", message);
   return new ResponseEntity(savedBooking, HttpStatus.CREATED);
}
```

At the last, in our Booking service, BookingService interface defined with all declaration of methods which are going to be implemented in BookingServiceImpl class which communicates with Database to store and retrieve data, also communicates to another service called as Payment Service to call it's APi in a synchronous way using restTemplate.

```
// Booking service Interface
public interface BookingService {

   public BookingInfoEntity makeBooking(BookingInfoEntity booking);

   public BookingInfoEntity getBookingBasedOnId(int id);

   public BookingInfoEntity updateBookingByCreatingTransaction(int id,
      BookingTransactionDTO bookingTransactionDTO);

   public void sendNotification(String topic, String key, String value) throws
      IOException;
}
```

These four methods are defined as follows.

The getBookingBasedOnID method takes id as an argument and finds the data into the database, if the data is present with the requested id then it returns the BookingInfoEntity instance, otherwise it throws an error that the booking id is not valid.

```
@Override
public BookingInfoEntity getBookingBasedOnId(int id) {
   // find booking from id into database raise error if id is not found
   if (bookingRepository.findById(id).isPresent()){
       return bookingRepository.findById(id).get();
   }
   throw new InvalidBooking( "Invalid Booking Id", 400 );
}
```

The validateTransaction method is used to validate the payment mode, if the payment mode is card/UPI then only API accepts the input otherwise it will throw an error that payment mode is not valid. And, if all validations are passed, it will find the Booking based on id in the database and returns the object.

```
public BookingInfoEntity validateTransaction(int id, BookingTransactionDTO
bookingTransactionDTO){
   // validate if paymentMode is not card or not upi then raise an error
   // return booking based on id from database if validation succeed
   if(bookingTransactionDTO.getPaymentMode() != null){
       String paymentMode =
bookingTransactionDTO.getPaymentMode().toLowerCase().strip();
       if(!(paymentMode.equalsIgnoreCase("card") ||
paymentMode.equalsIgnoreCase("upi"))){
           throw new InvalidTransaction( "Invalid mode of payment", 400 );
       }
   } else {
       throw new InvalidTransaction( "Invalid mode of payment", 400 );
   }

   BookingInfoEntity booking = getBookingBasedOnId(id);
   return booking;
}
```

The sendNotification method takes the topic name, key and value as an argument and sends the message to kafka topic. The other service named notification service fetches the data from the same topic later.

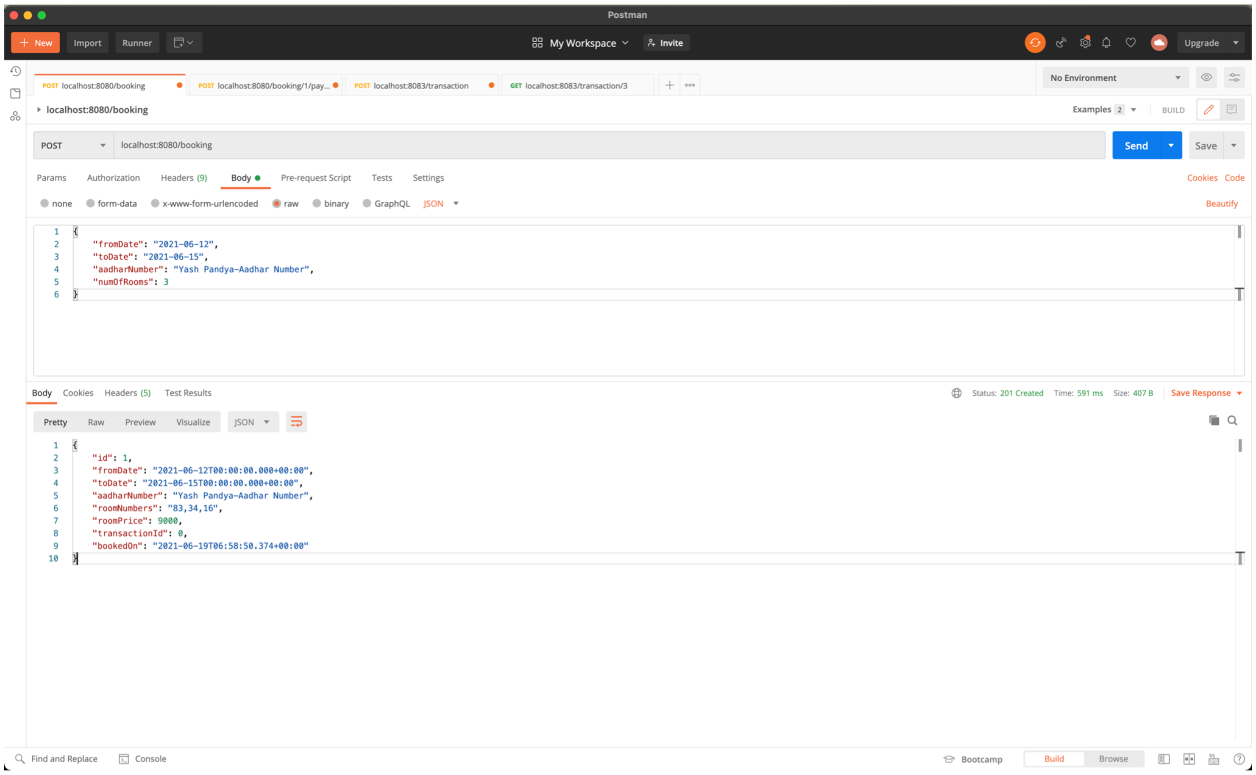
```
@Override
public void sendNotification(String topic, String key, String value) throws
IOException {
   // send notification to topic using kafka
   Producer<String, String> producer = kafkaConfig.getProducer();
   System.out.println(producer.metrics());
   // send single message
   producer.send(new ProducerRecord<String, String>(topic, key, value));
   producer.close();
   System.out.println("Notification sent");

}
```

The last method within the class is updateBookingByCreatingTransaction, this method mapped with */booking/{bookingId}/transaction* POST API. This method generates the payment Map and sends the data to the payment service using rest template and updates the transactionId back into BookingInfoEntity (booking table). Method also invokes the validation method to verify the payment mode before proceeding further.

```
@Override
public BookingInfoEntity updateBookingByCreatingTransaction(int id,
BookingTransactionDTO bookingTransactionDTO) {
   // create transaction data into payment service
```

```java
   BookingInfoEntity booking = validateTransaction(id, bookingTransactionDTO);

   Map<String,String> paymentUriMap = new HashMap<>();
   paymentUriMap.put("paymentMode", bookingTransactionDTO.getPaymentMode());
   paymentUriMap.put("bookingId",
String.valueOf(bookingTransactionDTO.getBookingId()));
   paymentUriMap.put("upiId", bookingTransactionDTO.getUpiId());
   paymentUriMap.put("cardNumber", bookingTransactionDTO.getCardNumber());

   // calling payment service API using rest template
   BookingTransactionDTO updateBookingTransactionDTO =
restTemplate.postForObject("http://PAYMENT-SERVICE/transaction", paymentUriMap,
BookingTransactionDTO.class);

   // return booking data with updated information
   if(updateBookingTransactionDTO != null){

booking.setTransactionId(updateBookingTransactionDTO.getTransactionId());
       bookingRepository.save(booking);
       return booking;
   }
   return null;
}
```

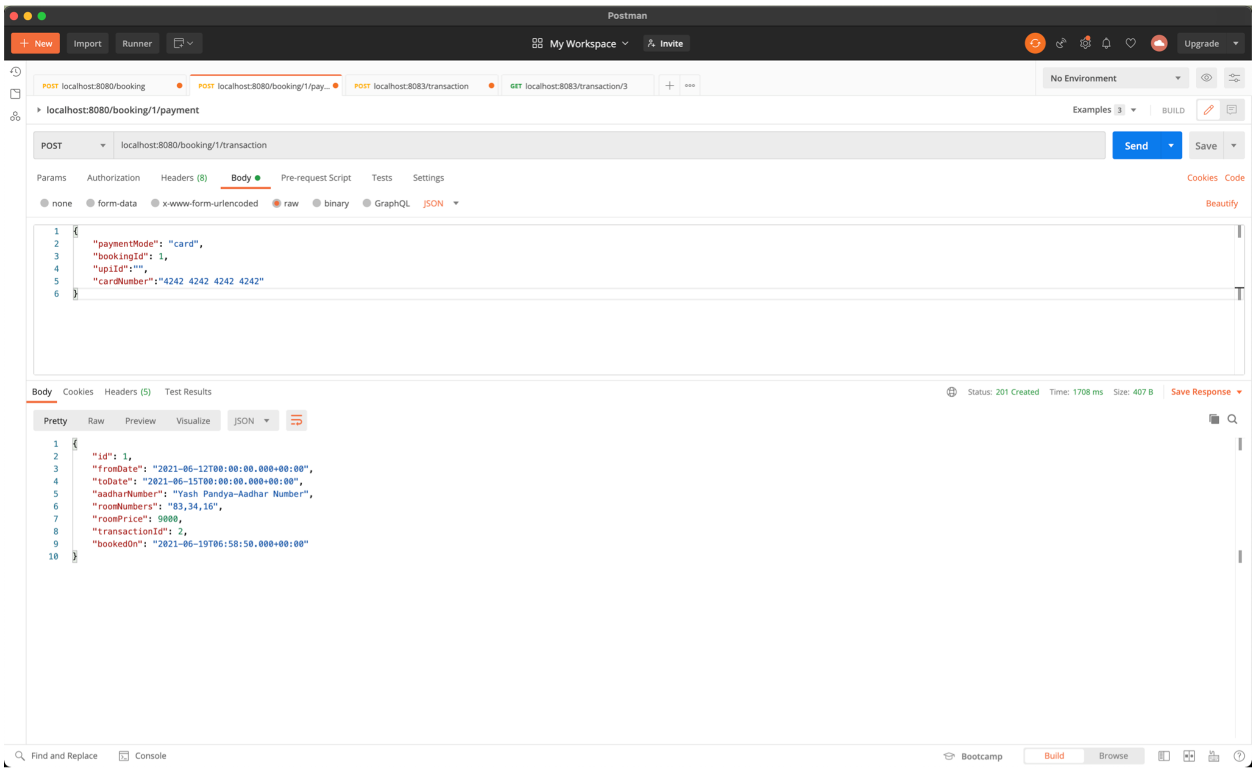
All the validation errors / exceptions are mapped in the exception directory and specific exception classes are implemented. InvalidBooking takes care of invalid booking id and InvalidTransaction takes care of payment mode invalidation errors.



POST localhost:8080/booking

```json
{
    "fromDate": "2021-06-12",
    "toDate": "2021-06-15",
    "aadharNumber": "Yash Pandya-Aadhar Number",
    "numOfRooms": 3
}
```

```json
{
    "id": 1,
    "fromDate": "2021-06-12T00:00:00.000+00:00",
    "toDate": "2021-06-15T00:00:00.000+00:00",
    "aadharNumber": "Yash Pandya-Aadhar Number",
    "roomNumbers": "83,34,16",
    "roomPrice": 9000,
    "transactionId": 0,
    "bookedOn": "2021-06-19T06:58:50.374+00:00"
}
```



POST localhost:8080/booking | POST localhost:8080/booking/1/pay... | POST localhost:8083/transaction | GET localhost:8083/transaction/3

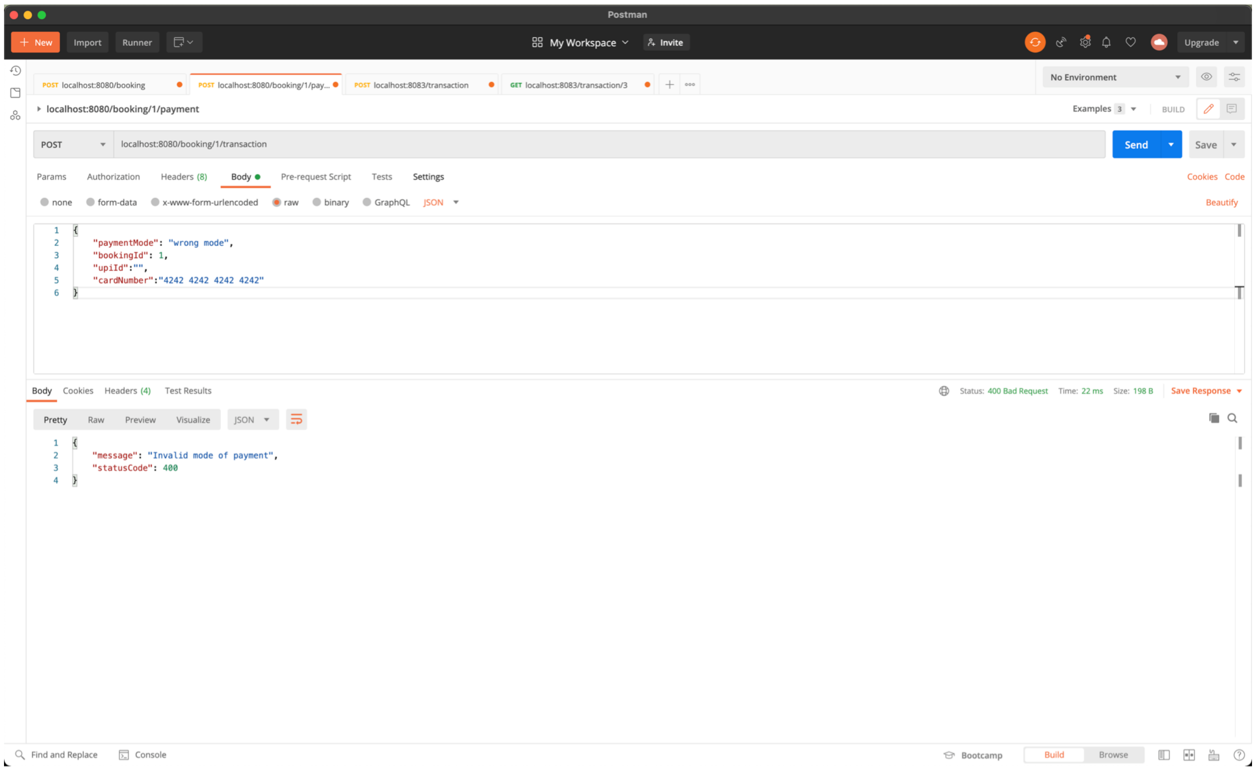
localhost:8080/booking/1/payment

POST localhost:8080/booking/1/transaction

```
{
    "paymentMode": "card",
    "bookingId": 1,
    "upiId":"",
    "cardNumber":"4242 4242 4242 4242"
}
```

Status: 201 Created Time: 1708 ms Size: 407 B

```
{
    "id": 1,
    "fromDate": "2021-06-12T00:00:00.000+00:00",
    "toDate": "2021-06-15T00:00:00.000+00:00",
    "aadharNumber": "Yash Pandya-Aadhar Number",
    "roomNumbers": "83,34,16",
    "roomPrice": 9000,
    "transactionId": 2,
    "bookedOn": "2021-06-19T06:58:50.000+00:00"
}
```



POST localhost:8080/booking | POST localhost:8080/booking/1/pay... | POST localhost:8083/transaction | GET localhost:8083/transaction/3

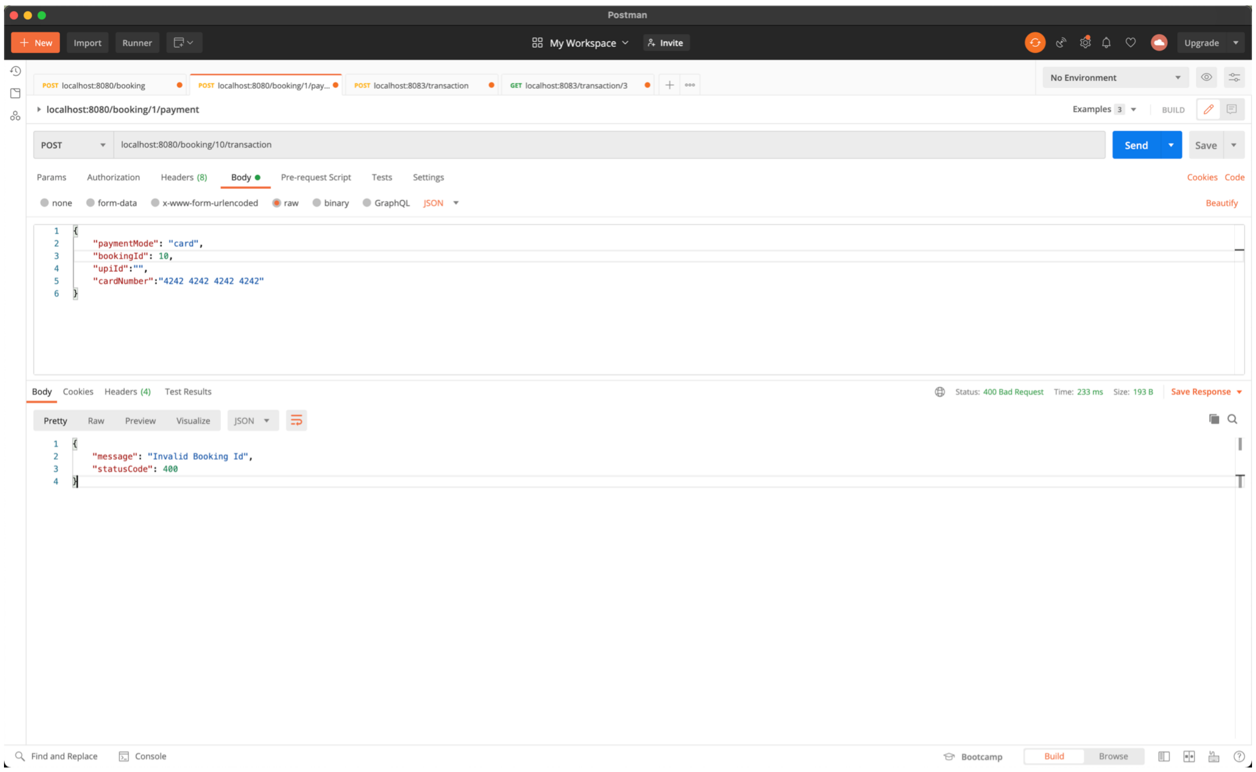
localhost:8080/booking/1/payment

POST localhost:8080/booking/1/transaction

```
{
    "paymentMode": "wrong mode",
    "bookingId": 1,
    "upiId":"",
    "cardNumber":"4242 4242 4242 4242"
}
```

Status: 400 Bad Request Time: 22 ms Size: 198 B

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



POST localhost:8080/booking/10/transaction

```
{
    "paymentMode": "card",
    "bookingId": 10,
    "upiId":"",
    "cardNumber":"4242 4242 4242 4242"
}
```

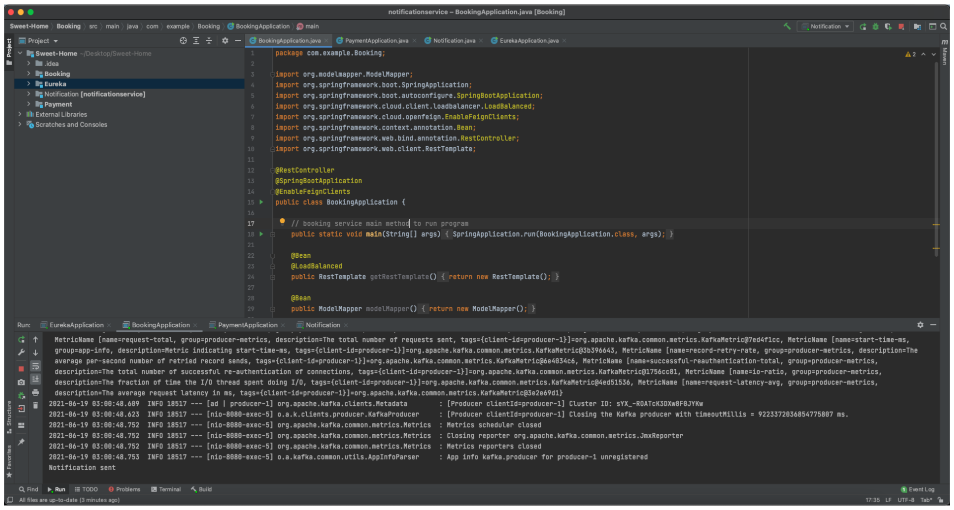
Status: 400 Bad Request   Time: 233 ms   Size: 193 B

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

NOTE:

There is variable name/parameter name mismatch in postman collection vs task description. bookingMode and paymentMode. paymentMode is used in my implementation.

- BookingService prints the message once notification is sent to kafka topic, the output is as shown below.



notificationservice – BookingApplication.java [Booking]

```java
package com.example.Booking;

import org.modelmapper.ModelMapper;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.context.annotation.Bean;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
@SpringBootApplication
@EnableFeignClients
public class BookingApplication {

    // booking service main method to run program
    public static void main(String[] args) { SpringApplication.run(BookingApplication.class, args); }

    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate() { return new RestTemplate(); }

    @Bean
    public ModelMapper modelMapper() { return new ModelMapper(); }
```

```
 MetricName [name=request-total, group=producer-metrics, description=The total number of requests sent, tags={client-id=producer-1}]=org.apache.kafka.common.metrics.KafkaMetric@7ed4f1cc, MetricName [name=start-time-ms,
 group=app-info, description=Metric indicating start-time-ms, tags={client-id=producer-1}]=org.apache.kafka.common.metrics.KafkaMetric@3b396643, MetricName [name=record-retry-rate, group=producer-metrics, description=The
 average per-second number of retried record sends, tags={client-id=producer-1}]=org.apache.kafka.common.metrics.KafkaMetric@6e4034c6, MetricName [name=successful-reauthentication-total, group=producer-metrics,
 description=The total number of successful re-authentication of connections, tags={client-id=producer-1}]=org.apache.kafka.common.metrics.KafkaMetric@1756cc81, MetricName [name=io-ratio, group=producer-metrics,
 description=The fraction of time the I/O thread spent doing I/O, tags={client-id=producer-1}]=org.apache.kafka.common.metrics.KafkaMetric@4ed51536, MetricName [name=request-latency-avg, group=producer-metrics,
 description=The average request latency in ms, tags={client-id=producer-1}]=org.apache.kafka.common.metrics.KafkaMetric@3e2e69d1}
2021-06-19 03:00:48.609  INFO 18517 --- [ad | producer-1] org.apache.kafka.clients.Metadata        : [Producer clientId=producer-1] Cluster ID: sYX_-ROATcK3DXw8F0JYKw
2021-06-19 03:00:48.623  INFO 18517 --- [nio-8080-exec-5] o.a.k.clients.producer.KafkaProducer     : [Producer clientId=producer-1] Closing the Kafka producer with timeoutMillis = 9223372036854775807 ms.
2021-06-19 03:00:48.752  INFO 18517 --- [nio-8080-exec-5] org.apache.kafka.common.metrics.Metrics  : Metrics scheduler closed
2021-06-19 03:00:48.752  INFO 18517 --- [nio-8080-exec-5] org.apache.kafka.common.metrics.Metrics  : Closing reporter org.apache.kafka.common.metrics.JmxReporter
2021-06-19 03:00:48.752  INFO 18517 --- [nio-8080-exec-5] org.apache.kafka.common.metrics.Metrics  : Metrics reporters closed
2021-06-19 03:00:48.753  INFO 18517 --- [nio-8080-exec-5] o.a.kafka.common.utils.AppInfoParser     : App info kafka.producer for producer-1 unregistered
Notification sent
```

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.

Based on the schema definitions of the transaction table, TransactionDetailsEntity class is created. The class is annotated with @Entity for the microservice to identify it as an entity and spring can create a table in the configured database for the same. The getter and setter methods along with toString are also created.

```
// transaction table entity class to map to and from database
@Entity(name="transaction")
public class TransactionDetailsEntity {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private int transactionId ;

   @Column
   private String paymentMode;

   @Column(nullable = false)
   private int bookingId;

   @Column(nullable = true)
   private String upiId ;

   @Column(nullable = true)
   private String cardNumber ;
}
```

Repository is an interface which extends JPA Repository, which persists Java Objects into relational databases.

```
@Repository
public interface PaymentRepository extends
JpaRepository<TransactionDetailsEntity, Integer> {
}
```

DTO class is independently worked as an interface among Repository and service layers. This is to keep the Database some portion of the back end isolated and unexposed.

TransactionDTO class is used to map the request/response for the /transaction POST API and /transaction/{transactionId} GET API.

```java
// DTO class which is used to map Payment detail
@JsonPropertyOrder({"id"})
public class TransactionDTO {

   @JsonProperty("id")
   private int transactionId ;

   private String paymentMode;

   private int bookingId;

   private String upiId ;

   private String cardNumber ;

   @JsonProperty("id")
   public int getTransactionId() {
       return transactionId;
   }

   @JsonProperty("transactionId")
   public void setTransactionId(int transactionId) {
       this.transactionId = transactionId;
   }
}
```

The ErrorDTO class is used to map the error/validation response.

```java
// DTO class which is used to map exception/error
public class ErrorDTO {

   private String message;

   private int statusCode ;

}
```

Payment controller is the interface between the presentation layer and the service class. This class has the POST methods which take the request input and send it to the service class and reverse. This class is also responsible for sending the response back in the HTTP form. it converts DTO to Entity and Entity back to DTO and sends either response or error back with appropriate status code.

```
// POST API to create transaction and store into Database
// It will return transaction/payment detail data as a response
@PostMapping(value="/transaction", consumes = MediaType.APPLICATION_JSON_VALUE,
       produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity createTransaction(@RequestBody TransactionDTO
transactionDTO){

   TransactionDetailsEntity transaction = modelMapper.map(transactionDTO,
TransactionDetailsEntity.class);
   TransactionDetailsEntity createtransaction =
paymentService.makeTransaction(transaction);
   TransactionDTO savedtransactionDTO = modelMapper.map(createtransaction,
TransactionDTO.class);

   return new ResponseEntity(savedtransactionDTO, HttpStatus.CREATED);

}


// GET API to get payment/transaction detail based on id
@GetMapping(value = "/transaction/{transactionId}")
public ResponseEntity getUser(@PathVariable(name = "transactionId")  int
transactionId){

   TransactionDetailsEntity transaction =
paymentService.getTransactionBasedOnId(transactionId);
   TransactionDTO transactionDTO =
POJOConvertor.covertPaymentEntityToDTO(transaction);
   return new ResponseEntity(transactionDTO, HttpStatus.OK);

}
```

At the last, in our Payment service, PaymentService interface defined with all declaration of methods which are going to be implemented in PaymentServiceImpl class which communicates with Database to store and retrieve data.

```
// Payment service Interface
public interface PaymentService {

   public TransactionDetailsEntity makeTransaction(TransactionDetailsEntity
payment);

   public TransactionDetailsEntity getTransactionBasedOnId(int id);

}
```

These two methods are defined as follows.

The getTransactionBasedOnId method takes id as an argument and finds the data into the database, if the data is present with the requested id then it returns the TransactionDetailsEntity instance, otherwise it throws an error that the payment id is not valid.

```
@Override
public TransactionDetailsEntity getTransactionBasedOnId(int id) {
    // find payment/transaction from id into database raise error if id is not
found
   if (paymentRepository.findById(id).isPresent()){
       return paymentRepository.findById(id).get();
   }
   throw new InvalidPayment( "Invalid Payment Id", 400 );
}
```

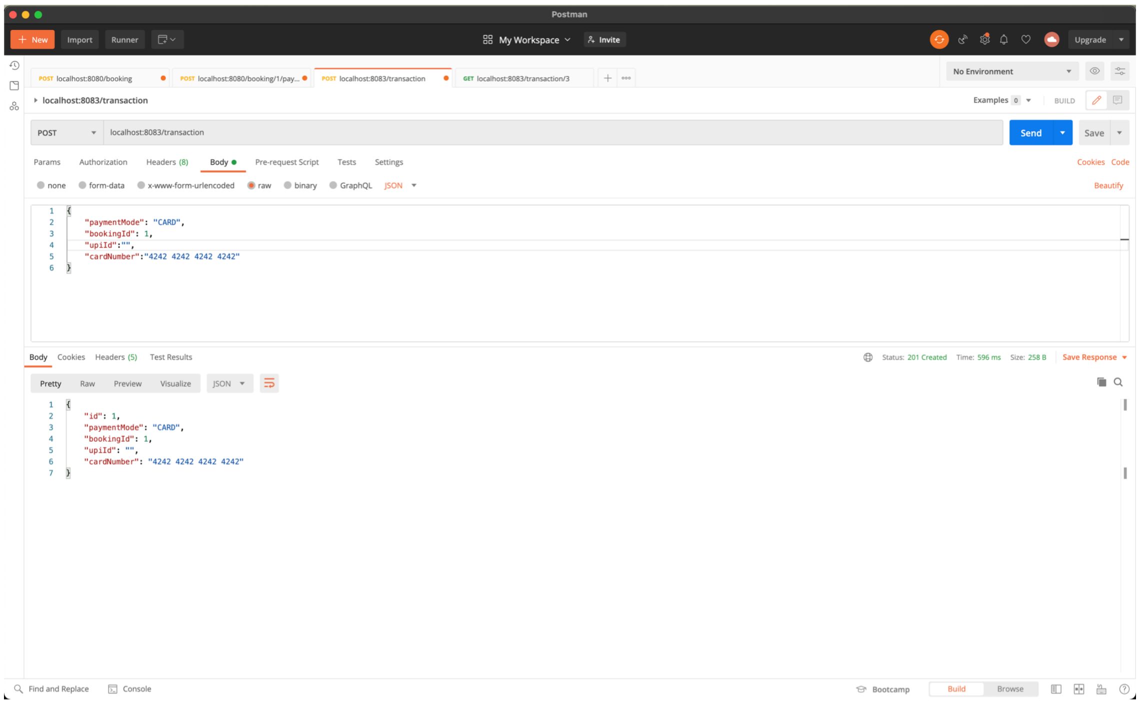
The second method within the class is makeTransaction, this method mapped with */transaction* POST API. This method stores the data into the database. Method also invokes the validation method to verify the payment mode before proceeding further.

```
@Override
public TransactionDetailsEntity makeTransaction(TransactionDetailsEntity
payment) {
   // validate if paymentMode is not card or not upi then raise an error
   // return payment based on id from database if validation succeed and
stores into the database
   if(payment.getPaymentMode() != null){
       String paymentMode = payment.getPaymentMode().toLowerCase().strip();
       if(!(paymentMode.equalsIgnoreCase("card") ||
paymentMode.equalsIgnoreCase("upi"))){
           throw new InvalidPaymentMode( "Invalid mode of payment", 400 );
       }
   } else {
       throw new InvalidPaymentMode( "Invalid mode of payment", 400 );
   }
   return paymentRepository.save(payment);
}
```

The other class under the utils directory is a POJOConvertor class contains two methods, convertPaymentEntityToDTO which converts the TransactionDetailsEntity into the TransactionDTO and the second method convertPaymentDTOToEntity converts the TransactionDTO to TransasctionDetailsEntity class.

```java
public static TransactionDTO covertPaymentEntityToDTO(TransactionDetailsEntity
payment) {
   // converts payment instance to paymentDTO instance
   TransactionDTO transactionDTO = new TransactionDTO();

   transactionDTO.setPaymentMode(payment.getPaymentMode());
   transactionDTO.setBookingId(payment.getBookingId());
   transactionDTO.setCardNumber(payment.getCardNumber());
   transactionDTO.setTransactionId(payment.getTransactionId());
   transactionDTO.setUpiId(payment.getUpiId());

   return transactionDTO;
}

public static TransactionDetailsEntity covertPaymentDTOToEntity(TransactionDTO
transactionDTO) {
   // converts paymentDTO instance to payment instance
   TransactionDetailsEntity payment = new TransactionDetailsEntity();

   payment.setPaymentMode(transactionDTO.getPaymentMode());
   payment.setBookingId(transactionDTO.getBookingId());
   payment.setCardNumber(transactionDTO.getCardNumber());
   payment.setTransactionId(transactionDTO.getTransactionId());
   payment.setUpiId(transactionDTO.getUpiId());

   return payment;
}
```

All the validation errors / exceptions are mapped in the exception directory and specific exception classes are implemented. InvalidPayment takes care of invalid payment/transaction id and InvalidPaymentMode takes care of payment mode invalidation errors.



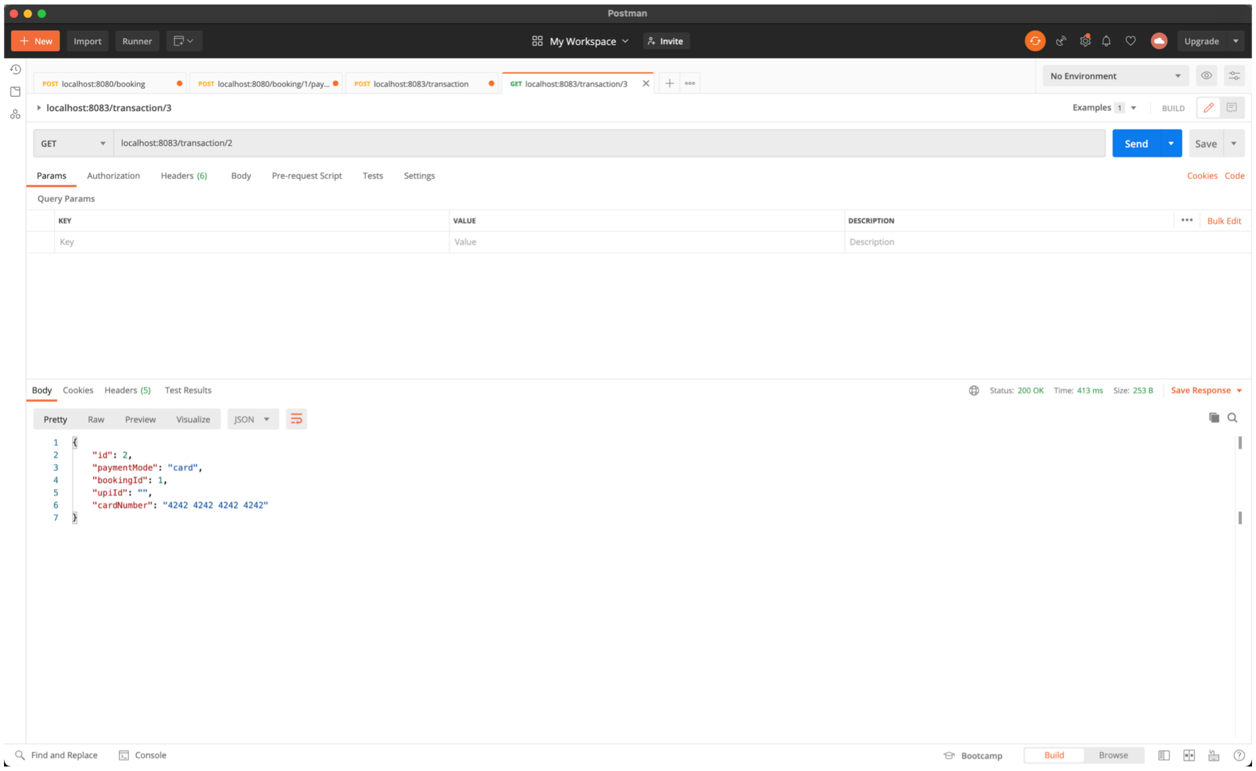
localhost:8083/transaction

POST localhost:8083/transaction

```
{
    "paymentMode": "CARD",
    "bookingId": 1,
    "upiId":"",
    "cardNumber":"4242 4242 4242 4242"
}
```

Status: 201 Created Time: 596 ms Size: 258 B

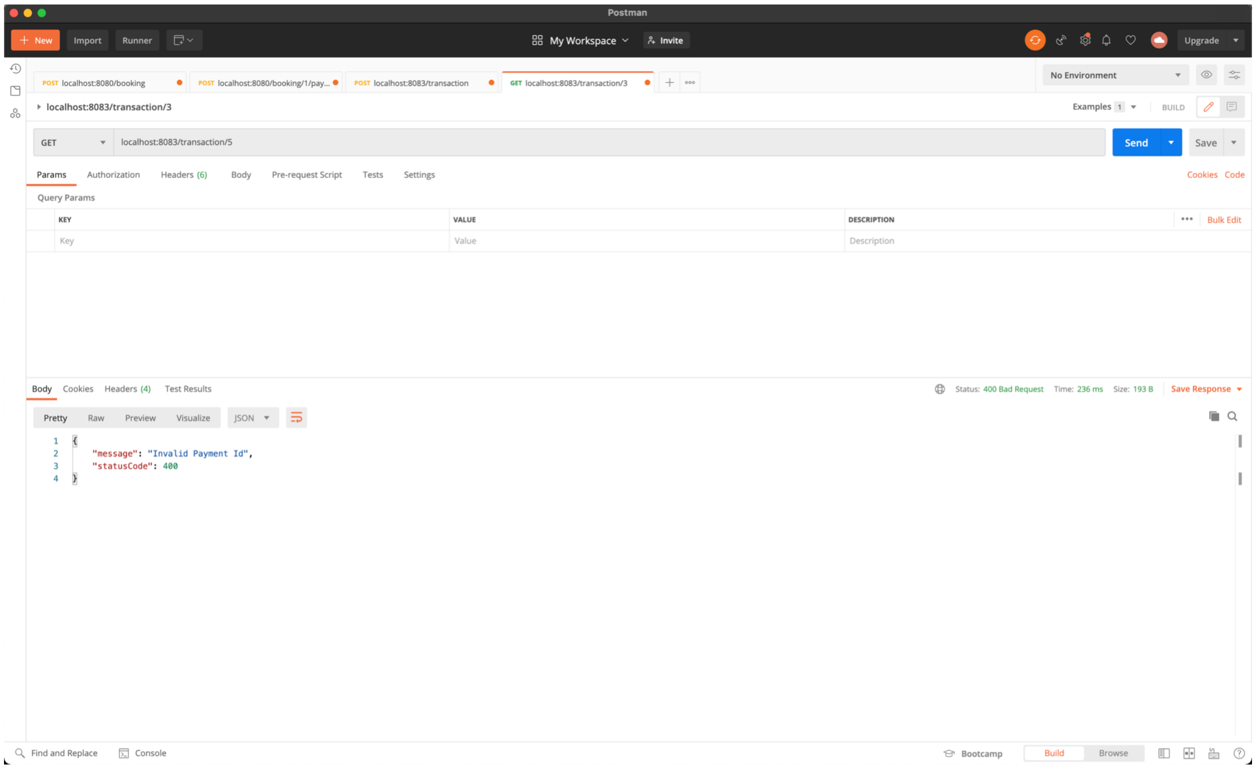
```
{
    "id": 1,
    "paymentMode": "CARD",
    "bookingId": 1,
    "upiId": "",
    "cardNumber": "4242 4242 4242 4242"
}
```



GET localhost:8083/transaction/2

Status: 200 OK Time: 413 ms Size: 253 B

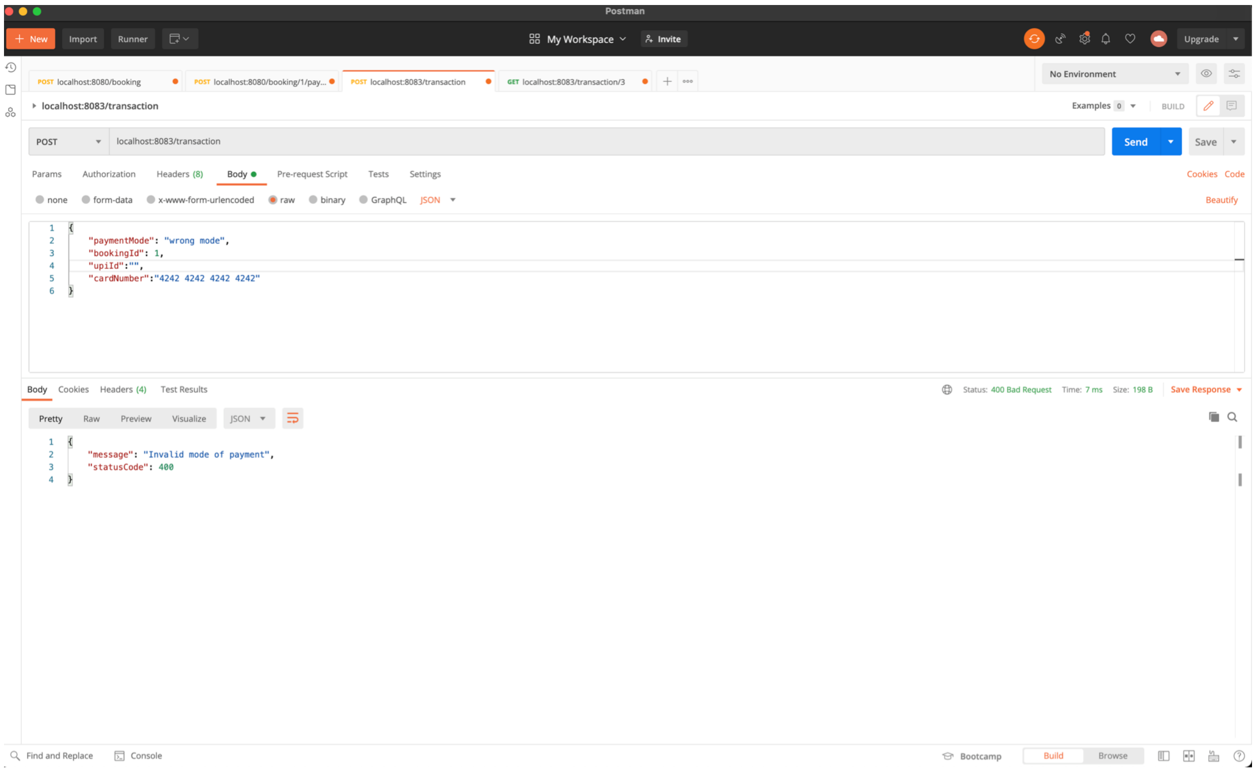
```
{
    "id": 2,
    "paymentMode": "card",
    "bookingId": 1,
    "upiId": "",
    "cardNumber": "4242 4242 4242 4242"
}
```



GET localhost:8083/transaction/5

Status: 400 Bad Request Time: 236 ms Size: 193 B

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



POST localhost:8083/transaction

```
{
    "paymentMode": "wrong mode",
    "bookingId": 1,
    "upiId":"",
    "cardNumber":"4242 4242 4242 4242"
}
```

Status: 400 Bad Request Time: 7 ms Size: 198 B

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

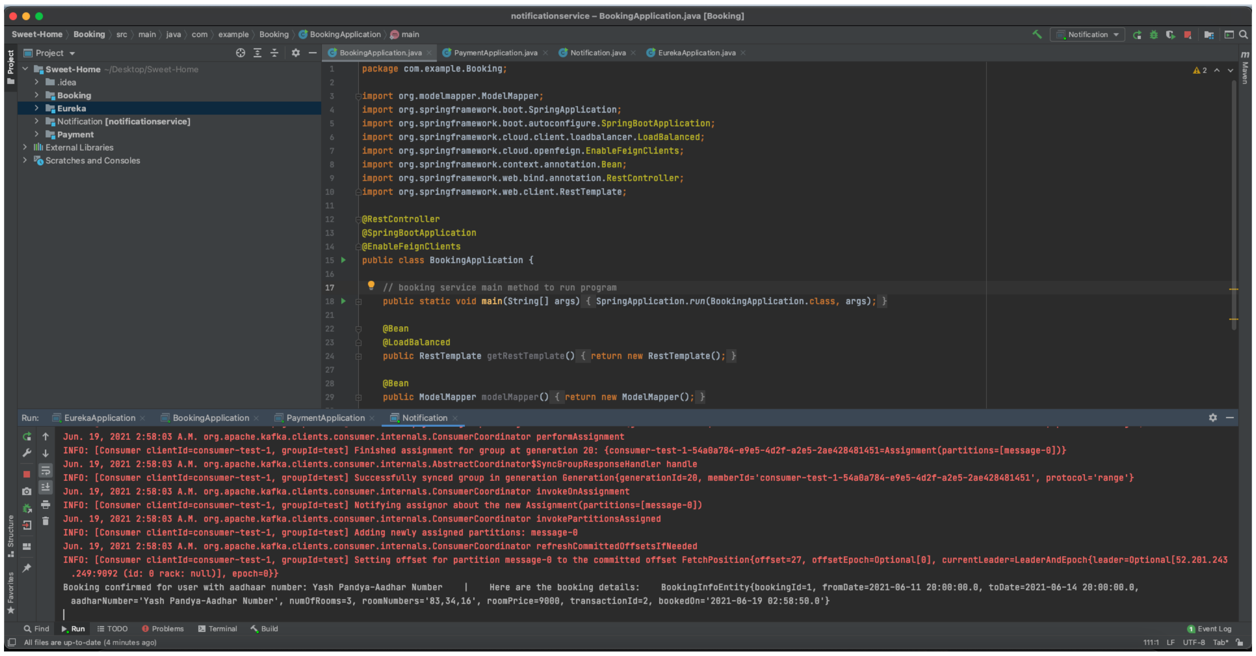
<u>Notification Service</u>:

This service is responsible for fetching the message from kafka topic and prints on the console. This is not a spring boot project instead, it is a maven based simple java project.

The main class Notification acts as a kafka consumer and fetch the message from the configured kafka topic.

```
public class Notification {

   public static void main(String[] args) {

       Properties props = new Properties();
       props.setProperty("bootstrap.servers",
            "ec2-174-129-48-56.compute-1.amazonaws.com:9092");
       props.setProperty("group.id", "test");
       props.setProperty("enable.auto.commit", "true");
       props.setProperty("auto.commit.interval.ms", "1000");
```

```
        props.setProperty("key.deserializer",
             "org.apache.kafka.common.serialization.StringDeserializer");
        props.setProperty("value.deserializer",
             "org.apache.kafka.common.serialization.StringDeserializer");
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("message"));

        try {
            while (true) {
                ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
                for (ConsumerRecord<String, String> record : records)
                    System.out.printf("%s%n", record.value());
            }
        } finally {
            consumer.close();
        }
    }
}
```



```
package com.example.Booking;

import org.modelmapper.ModelMapper;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.context.annotation.Bean;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
@SpringBootApplication
@EnableFeignClients
public class BookingApplication {

    // booking service main method to run program
    public static void main(String[] args) { SpringApplication.run(BookingApplication.class, args); }

    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate() { return new RestTemplate(); }

    @Bean
    public ModelMapper modelMapper() { return new ModelMapper(); }
```

```
Jun. 19, 2021 2:58:03 A.M. org.apache.kafka.clients.consumer.internals.ConsumerCoordinator performAssignment
INFO: [Consumer clientId=consumer-test-1, groupId=test] Finished assignment for group at generation 20: {consumer-test-1-54a0a784-e9e5-4d2f-a2e5-2ae428481451=Assignment(partitions=[message-0])}
Jun. 19, 2021 2:58:03 A.M. org.apache.kafka.clients.consumer.internals.AbstractCoordinator$SyncGroupResponseHandler handle
INFO: [Consumer clientId=consumer-test-1, groupId=test] Successfully synced group in generation Generation{generationId=20, memberId='consumer-test-1-54a0a784-e9e5-4d2f-a2e5-2ae428481451', protocol='range'}
Jun. 19, 2021 2:58:03 A.M. org.apache.kafka.clients.consumer.internals.ConsumerCoordinator invokeOnAssignment
INFO: [Consumer clientId=consumer-test-1, groupId=test] Notifying assignor about the new Assignment(partitions=[message-0])
Jun. 19, 2021 2:58:03 A.M. org.apache.kafka.clients.consumer.internals.ConsumerCoordinator invokePartitionsAssigned
INFO: [Consumer clientId=consumer-test-1, groupId=test] Adding newly assigned partitions: message-0
Jun. 19, 2021 2:58:03 A.M. org.apache.kafka.clients.consumer.internals.ConsumerCoordinator refreshCommittedOffsetsIfNeeded
INFO: [Consumer clientId=consumer-test-1, groupId=test] Setting offset for partition message-0 to the committed offset FetchPosition{offset=27, offsetEpoch=Optional[0], currentLeader=LeaderAndEpoch{leader=Optional[52.201.243
 .249:9092 (id: 0 rack: null)], epoch=0}}
Booking confirmed for user with aadhaar number: Yash Pandya-Aadhar Number    |    Here are the booking details:    BookingInfoEntity{bookingId=1, fromDate=2021-06-11 20:00:00.0, toDate=2021-06-14 20:00:00.0,
 aadharNumber='Yash Pandya-Aadhar Number', numOfRooms=3, roomNumbers='83,34,16', roomPrice=9000, transactionId=2, bookedOn='2021-06-19 02:58:50.0'}
```

<u>Eureka</u>:

This service is responsible for the service registry of the project based on spring boot dependencies. The main class is annotated with @EnableEurekaServer, Register-with-Eureka and fetch-registry is set to "false" so that it does not get included into the registry. The Eureka server can be accessed on localhost port 8761.

Service registry (Eureka) configuration is written in yml file named as application.yml

```
server:
 port: 8761

eureka:
 client:
   register-with-eureka: false
   fetch-registry: false
```

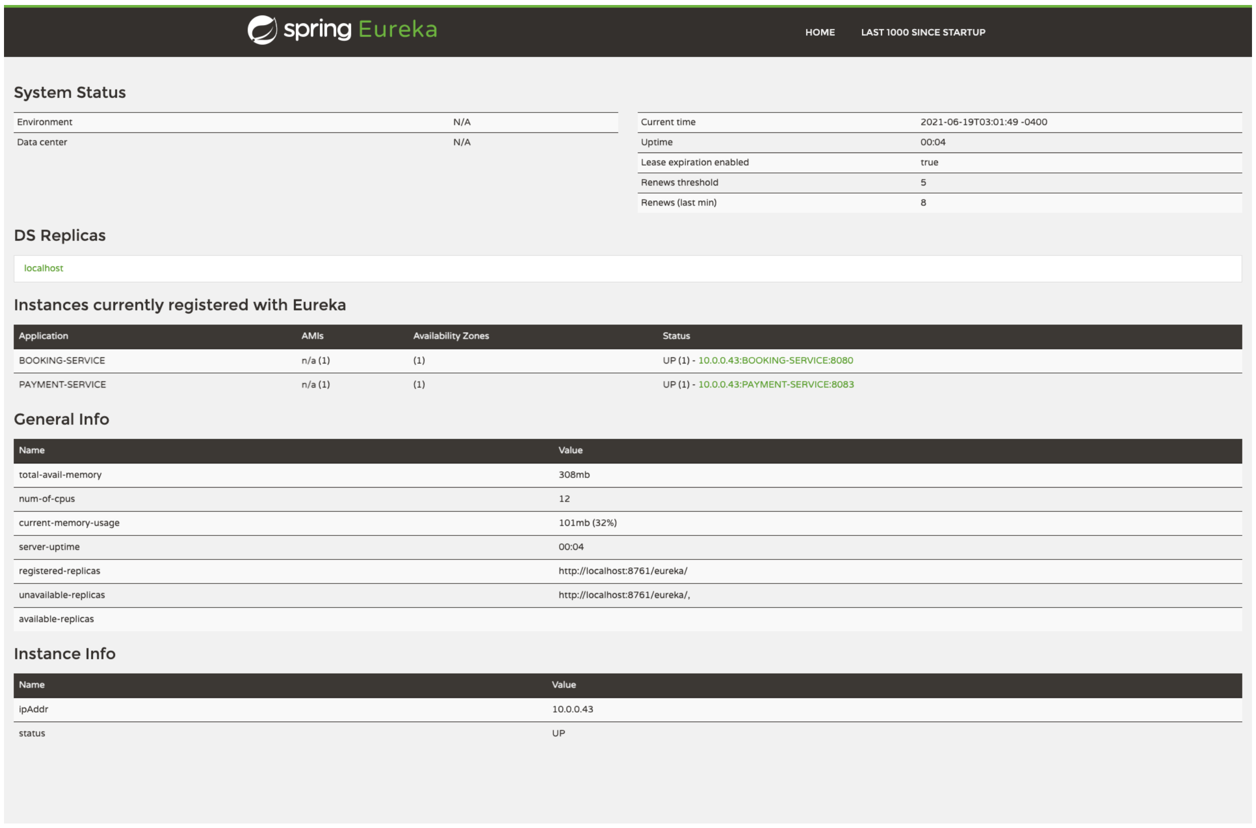
The BookingService registry configuration is as below. Which states that Booking service runs on port 8080

```
server:
 port: 8080

spring:
 application:
   name: BOOKING-SERVICE

eureka:
 client:
   register-with-eureka: true
   fetch-registry: true
   service-url:
     defaultZone: http://localhost:8761/eureka/
   instance:
     hostname: localhost
```

The PaymentService registry configuration is as below. Which states that Booking service runs on port 8083

```
server:
 port: 8083

spring:
 application:
   name: PAYMENT-SERVICE

eureka:
 client:
   register-with-eureka: true
```

```
  fetch-registry: true
  service-url:
    defaultZone: http://localhost:8761/eureka/
  instance:
    hostname: localhost
```



System Status

Environment	N/A
Data center	N/A

Current time	2021-06-19T03:01:49 -0400
Uptime	00:04
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) - 10.0.0.43:BOOKING-SERVICE:8080
PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - 10.0.0.43:PAYMENT-SERVICE:8083

General Info

Name	Value
total-avail-memory	308mb
num-of-cpus	12
current-memory-usage	101mb (32%)
server-uptime	00:04
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,
available-replicas	

Instance Info

Name	Value
ipAddr	10.0.0.43
status	UP

- All other configurations related to each application can be found in application.properties file.

How to Run:

- Start/run Eureka application
- Update the database URI in application.properties, update ec2 instance URI in KafkaConfig class and run Booking service application
- Update the database URL in application.properties and run Payment service application
- Update ec2 instance URI in Main class of Notification service and run the Notification service application