


# CODING LOGIC

## \*SDK setting of this code:

SDK:	 17 Oracle OpenJDK version 17.0.6 ▼	<button>Edit</button>
------	--	-----------------------

## \*Endpoints for booking service

- POST: <http://localhost:8081/hotel/booking>
- POST: <http://localhost:8081/hotel/booking/1/transaction>

## \*Endpoints for payment service

- POST: <http://localhost:8083/payment/transaction>
- GET: <http://localhost:8083/payment/transaction/1>

## 1. Eureka Server

- Adding the dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

- Enabling Eureka Server in the application with Eureka Server dependency

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaserverApplication {

    public static void main(String[] args) { SpringApplication.run(EurekaserverApplication.class, args); }

}
```

- Adding application.yml file in the resources for eureka server configuration

```
server:
  port: 8761

#not register the server itself
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

- The port of eureka server is 8761.
- The code is configuring eureka server to prevent the server from registering itself.

## 2. Booking Service Application

- H2 Database URL: `http://localhost:8081/h2-console`
- BookingInfoEntity in the model package is used to interact with database. It will include the following properties:

```

1  {
2      "id": 1,
3      "fromDate": "2021-06-10T00:00:00.000+00:00",
4      "toDate": "2021-06-15T00:00:00.000+00:00",
5      "aadharNumber": "Sample-Aadhar-Number",
6      "numOfRooms": 3,
7      "roomNumbers": "70,76,26",
8      "roomPrice": 15000,
9      "transactionId": 2,
10     "bookedOn": "2023-05-01T06:16:29.527+00:00"
11 }

```

- PaymentInfo is used to receive the request from users for Payment process and is located in *dto* package.
  - It will pass the data to the subsequent POST request from Booking Service to Payment service to get the transaction Id.
  - It will include the following properties:

```

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

```

- BookingRepository interface in the *repository* package defines the contract for data access operations.
- BookingServiceImpl will implement the interface Booking Service. Both classes are in *service* package. BookingServiceImpl will override the following methods and need the following dependencies:
  - BookingRepository instance is used to interact with data access layer.
  - RestTemplate is used to make Http requests for Payment Service.
  - saveBooking(BookingInfoDto bookingInfoDto): after receive the request from user from the Controller layer, this method will save the provided information to data base and return the saved object. The following is how the data provided by the user is saved to the database:

1. *Creating an object to be saved in the database*

```
BookingInfoEntity bookingToSave = new BookingInfoEntity();
```

2. *Randomly generating room numbers in accordance with the number of rooms that the user requests. Then, the array of room numbers will be converted to String type.*

```
ArrayList roomNumsResult = getRandomNumbers(bookingInfoDto.getNumOfRooms());  
String roomNums = String.join( delimiter: ",", roomNumsResult);
```

3. *Creating the current time, which will be used in the setBookedOn method.*

```
Date currentDate = new Date(System.currentTimeMillis());
```

4. *Getting the fromDate and toDate value from the bookingInfoDto provided by the user and calculating the price based on numbers of rooms, number of days and unit price*

```
// Room Price Calculation  
long fromDate = bookingInfoDto.getFromDate().getTime();  
long toDate = bookingInfoDto.getToDate().getTime();  
int numOfDay = (int) ((toDate - fromDate)/(1000 * 60 * 60 * 24));  
int price = 1000*bookingInfoDto.getNumOfRooms()*numOfDay;
```

5. *Fulfilling all the necessary properties in the bookingToSave object and saving this object to the database whose object output will be returned in this "saveBooking" method. At this point, the default value of transactionId is 0.*

```
bookingToSave.setFromDate(bookingInfoDto.getFromDate());
bookingToSave.setToDate(bookingInfoDto.getToDate());
bookingToSave.setAadharNumber(bookingInfoDto.getAadharNumber());
bookingToSave.setNumOfRooms(bookingInfoDto.getNumOfRooms());
bookingToSave.setRoomPrice(price);
bookingToSave.setRoomNumbers(roomNums);
bookingToSave.setTransactionId(0);
bookingToSave.setBookedOn(currentDate);

BookingInfoEntity savedBooking = bookingRepository.save(bookingToSave);

return savedBooking;
```

- takingPayment(int bookingId, PaymentInfo paymentInfo): This method will check if the mode of payment and booking id are valid. If all the conditions are met, this method will return the booking information object updated with transactionId. If the conditions are not satisfied, it will throw an exception. The following is the code flow in this method:
  1. *Checking if the mode of payment and booking id provided by the user is valid. If the conditions are not satisfied, the code will throw exception with the respective message and status code.*

```
// exception will be thrown when the user send the wrong mode of payment
if (!inputPaymentMethod.equalsIgnoreCase(MODE_UPI) && !inputPaymentMethod.equalsIgnoreCase(MODE_CARD)) {
    throw new InvalidPaymentInfoException(HttpStatus.BAD_REQUEST, "Invalid mode of payment");
}

BookingInfoEntity savedBooking = bookingRepository.findById(bookingId)
    .orElseThrow(() -> new InvalidPaymentInfoException(HttpStatus.BAD_REQUEST, " Invalid Booking Id"));
```

2. As all the validations are done, the below code will be executed. The code is making a POST request to Payment service using Rest Template to get the transactionId. This request will also save the user payment information to the database in Payment service.

```
PaymentInfo transaction = new PaymentInfo();
transaction.setBookingId(bookingId);
transaction.setPaymentMode(paymentInfo.getPaymentMode());
transaction.setCardNumber(paymentInfo.getCardNumber());
transaction.setUpiId(paymentInfo.getUpiId());

HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

// send a post request to the payment service to get transaction info
try {
    int transactionId = restTemplate.postForObject(postPaymentAppUrl, new HttpEntity<>(transaction, headers), Integer.class);
    savedBooking.setTransactionId(transactionId);

    // if the communication works, this message will be printed on the console.
    String message = "Booking confirmed for user with aadhaar number: "
        + savedBooking.getAadhaarNumber()
        + " | "
        + "Here are the booking details: " + savedBooking.toString();

    System.out.println(message);
} catch (RestClientException e) {

    throw new InvalidPaymentInfoException(HttpStatus.BAD_REQUEST, " Invalid Booking Id ");
}
```

3. The code proceeds to updating the current booking formation object with the retrieved transactionId, which returns the updatedBookingWithTransactionId. Finally, updatedBookingWithTransactionId will be returned to the user.

```
savedBooking.setTransactionId(transactionId);
```

```
BookingInfoEntity updatedBookingWithTransactionId = bookingRepository.save(savedBooking);
```

```
return updatedBookingWithTransactionId;
```

- getRandomNumbers(int count): this method is used to generate the room numbers in accordance with the required number of rooms from the user. It will return the array of randomly generated numbers.

1 usage

```
public static ArrayList<String> getRandomNumbers(int count){  
    Random rand = new Random();  
    int upperBound = 100;  
    ArrayList<String>numberList = new ArrayList<>();  
  
    for (int i=0; i<count; i++){  
        numberList.add(String.valueOf(rand.nextInt(upperBound)));  
    }  
  
    return numberList;  
}
```

#### - Booking Controller

- Post Mapping method with “/booking” uri: This method will get a BookingInfoEntity object in the request body from the user and communicate with the service layer to save the provided data to the database. When the data is successfully saved, the user will receive a BookingInfoEntity object and Http status “Created”.

```
@PostMapping("/booking")  
public ResponseEntity<BookingInfoEntity> createBooking(@RequestBody BookingInfoDto bookingInfoDto) {  
  
    BookingInfoEntity savedBooking = bookingService.saveBooking(bookingInfoDto);  
  
    return new ResponseEntity<>(savedBooking, HttpStatus.CREATED);  
}
```

- Post Mapping method with ““/booking/{bookingId}/transaction” uri: this method will get a PaymentInfo object in the request body and extract the bookingId in the endpoint from the user. It will communicate with the service layer to validate the mode of payment and bookingId and then proceed to make a request to save the provided data to the database in the Payment service. The service layer will return a BookingInfoEntity object updated with the transactionId. Then, a message for successful payment information process is printed in the console. Finally, the returned BookingInfoEntity is sent to the user.

```
// creating payment information
no usages
@PostMapping(value = "booking/{bookingId}/transaction", produces = MediaType.APPLICATION_JSON_VALUE, consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<BookingInfoEntity> requestPaymentService(@PathVariable(name = "bookingId") int bookingId , @RequestBody PaymentInfo paymentInfo) {
    BookingInfoEntity bookingInfo = bookingService.takingPayment(bookingId, paymentInfo);
    return new ResponseEntity<>(bookingInfo, HttpStatus.CREATED);
}
```

- 
- ModeOfPayment interface will define the constant value for the modes of payment.

```
2 usages
public interface ModeOfPayment {
    2 usages
    String MODE_UPI = "UPI";
    2 usages
    String MODE_CARD = "CARD";
}
```

- Exception Handling:
  - ErrorDetails class: This class will formulate an object having a message and a status code property. This object will be returned to the user when there is an exception thrown.
  - InvalidPaymentException class: this is a custom exception handling class. This class extends RuntimeException and accept status and message input. This custom exception will be thrown in the case of invalid mode of payment or invalid booking id.

- GlobalExceptionHandler class: this class will handle all the exceptions from the application. It is annotated with `@ControllerAdvice` and extends `ResponseBodyExceptionHandler`. To handle custom exception “`InvalidPaymentInfoException`”, the class is specified in the attribute of `@ExceptionHandler()` annotation. Inside the exception handling method, an instance of `ErrorDetails` is created with exception message and Http status code value passed in. The method will return this instance and a Http status.

```
no usages
@ControllerAdvice
public class GlobalExceptionHandler extends ResponseExceptionHandler {

    no usages
    @ExceptionHandler({InvalidPaymentInfoException.class})
    public ResponseEntity<ErrorDetails> handleBlogAPIException(InvalidPaymentInfoException exception, WebRequest webRequest) {
        ErrorDetails errorDetails = new ErrorDetails(exception.getMessage(), exception.getStatus().value());
        return new ResponseEntity<>(errorDetails, HttpStatus.BAD_REQUEST);
    }
}
```

### 3. Payment Service Application

- H2 Database URL: <http://localhost:8083/h2-console>
- `TransactionDetailsEntity` class in the `model` package: This class will be used when the application is interacting with the database. It has the following properties:

```
{
  "transactionId": 1,
  "paymentMode": "UPI",
  "bookingId": 1,
  "upiId": "card details",
  "cardNumber": ""
}
```

- `PaymentRepository` interface in the `repository` package defines the contract for data access operations.
- `TransactionServiceImpl` class will implement `TransactionService` interface. Both classes are in `service` package. `PaymentServiceImpl` will override the following methods and need the following dependencies:
  - `PaymentRepository` instance is used to interact with data access layer.



- creatingTransaction(TransactionDetailsEntity transactionDetails): this method will save the passed-in parameter to the data base and return the transactionId.

```
@Override
public int creatingTransaction(TransactionDetailsEntity transactionDetails) {

    TransactionDetailsEntity transactionToSave = new TransactionDetailsEntity();

    transactionToSave.setBookingId(transactionDetails.getBookingId());
    transactionToSave.setCardNumber(transactionDetails.getCardNumber());
    transactionToSave.setUpiId(transactionDetails.getUpiId());
    transactionToSave.setPaymentMode(transactionDetails.getPaymentMode());

    TransactionDetailsEntity savedTransaction = paymentRepository.save(transactionToSave);

    return savedTransaction.getTransactionId();
}
```

- creatingTransaction(TransactionDetailsEntity transactionDetails): this method will get the TransactionDetailsEntity object from the database and return it to the user.

```
1 usage
@Override
public TransactionDetailsEntity gettingTransaction(int transactionId) {

    TransactionDetailsEntity transactionDetails = paymentRepository.findById(transactionId).get();

    return transactionDetails;
}
```