PG FSD: API End Points and Communication

Source code:

1. **Booking Microservice**

```
// BookingController.java
@RestController
@RequestMapping("/bookings")
public class BookingController {

    @Autowired
    private BookingService bookingService;

    @PostMapping
    public ResponseEntity<Booking>
createBooking(@RequestBody Booking booking) {
        Booking createdBooking =
bookingService.createBooking(booking);
        return new ResponseEntity<>(createdBooking,
HttpStatus.CREATED);
    }

    // Other CRUD operations and endpoints
}

// BookingService.java
@Service
public class BookingService {

    @Autowired
    private BookingRepository bookingRepository;

    public Booking createBooking(Booking booking) {
        // Perform validation and business logic
        return bookingRepository.save(booking);
    }
```

```
    // Other service methods
}

// Booking.java - Entity class
@Entity
public class Booking {
    // Fields, getters, setters
}

// BookingRepository.java - JPA Repository
@Repository
public interface BookingRepository extends
JpaRepository<Booking, Long> {
    // Additional query methods if needed
}
```

2. Passenger Microservice

```
// PassengerController.java
@RestController
@RequestMapping("/passengers")
public class PassengerController {

    @Autowired
    private PassengerService passengerService;

    @PostMapping
    public ResponseEntity<Passenger>
createPassenger(@RequestBody Passenger passenger) {
        Passenger createdPassenger =
passengerService.createPassenger(passenger);
        return new ResponseEntity<>(createdPassenger,
HttpStatus.CREATED);
    }
```

```
    // Other CRUD operations and endpoints
}

// PassengerService.java
@Service
public class PassengerService {

    @Autowired
    private PassengerRepository passengerRepository;

    public Passenger createPassenger(Passenger passenger) {
        // Perform validation and business logic
        return passengerRepository.save(passenger);
    }

    // Other service methods
}

// Passenger.java - Entity class
@Entity
public class Passenger {
    // Fields, getters, setters
}

// PassengerRepository.java - JPA Repository
@Repository
public interface PassengerRepository extends
JpaRepository<Passenger, Long> {
    // Additional query methods if needed
}
```

In this example, PassengerService and BookingService handle the business logic, while PassengerController and BookingController handle HTTP requests and responses. The entity classes (Passenger

and Booking) represent the data model, and JPA repositories (PassengerRepository and BookingRepository) provide data access operations. Make sure to configure your Spring Boot application class and application properties appropriately for database connections and other configurations. Additionally, don't forget to include dependencies for Spring Boot, Spring Data JPA, and any other required libraries in your pom.xml or build.gradle file.

—----------

3. Payment Service:

Responsibilities:

Process payments for bookings.

Implementation:

Develop a separate Spring Boot project for the Payment Service.

Integrate with a payment gateway (e.g., Stripe, PayPal) for processing payments.

Implement endpoints for handling payment requests and callbacks.

Testing:

Write unit tests to ensure payment processing logic works correctly.

API Design:

Design RESTful APIs for initiating and processing payments.

4. Notification Service:

Responsibilities:

Send notifications to passengers regarding bookings and other relevant information.

Implementation:

Create another Spring Boot project for the Notification Service.

Implement messaging functionality using a message broker (e.g., RabbitMQ, Kafka).

Set up event listeners to handle booking-related events and send notifications accordingly.

Testing:

Write unit tests for notification sending logic.

API Design:

Design RESTful APIs for managing notification preferences and settings.

```
// PaymentController.java
@RestController
@RequestMapping("/payments")
public class PaymentController {

    @Autowired
    private PaymentService paymentService;

    @PostMapping("/process")
    public ResponseEntity<String> processPayment(@RequestBody
PaymentRequest paymentRequest) {
        String paymentStatus =
paymentService.processPayment(paymentRequest);
        return new ResponseEntity<>(paymentStatus, HttpStatus.OK);
    }

    // Other endpoints for handling payment callbacks, etc.
}

// PaymentService.java
@Service
public class PaymentService {

    @Autowired
    private PaymentGateway paymentGateway;

    public String processPayment(PaymentRequest paymentRequest) {
        // Call the payment gateway API to process the payment
        return paymentGateway.processPayment(paymentRequest);
    }

    // Other service methods
```

```
}

// PaymentGateway.java (Interface)
public interface PaymentGateway {
    String processPayment(PaymentRequest paymentRequest);
}

// PaymentRequest.java
public class PaymentRequest {
    // Define payment request attributes
}

// PaymentGatewayImpl.java (Example implementation using Stripe)
@Component
public class StripePaymentGateway implements PaymentGateway {

    @Override
    public String processPayment(PaymentRequest paymentRequest) {
        // Implement payment processing logic using Stripe API
        return "Payment processed successfully";
    }
}
```

4. Notification Service

```
// NotificationController.java
@RestController
@RequestMapping("/notifications")
public class NotificationController {

    @Autowired
    private NotificationService notificationService;

    @PostMapping("/send")
    public ResponseEntity<String> sendNotification(@RequestBody
NotificationRequest notificationRequest) {
```

```
        String notificationStatus =
notificationService.sendNotification(notificationRequest);
        return new ResponseEntity<>(notificationStatus, HttpStatus.OK);
    }

    // Other endpoints for managing notification preferences, settings, etc.
}

// NotificationService.java
@Service
public class NotificationService {

    @Autowired
    private MessageBroker messageBroker;

    public String sendNotification(NotificationRequest notificationRequest) {
        // Publish notification message to the message broker
        messageBroker.publishMessage(notificationRequest);
        return "Notification sent successfully";
    }

    // Other service methods
}

// MessageBroker.java (Interface)
public interface MessageBroker {
    void publishMessage(NotificationRequest notificationRequest);
}

// MessageBrokerImpl.java (Example implementation using RabbitMQ)
@Component
public class RabbitMQMessageBroker implements MessageBroker {

    @Override
    public void publishMessage(NotificationRequest notificationRequest) {
```

```
        // Implement logic to publish message to RabbitMQ
    }
}
```

In this example, PaymentService and NotificationService handle the business logic, while PaymentController and NotificationController handle HTTP requests and responses. The PaymentGateway and MessageBroker interfaces provide abstraction for integrating with payment gateways and message brokers, respectively. StripePaymentGateway and RabbitMQMessageBroker are example implementations for Stripe payment gateway integration and RabbitMQ message broker integration, respectively.

Make sure to configure your Spring Boot application class and application properties appropriately. Additionally, include dependencies for Spring Boot, Spring Web, and any other required libraries in your pom.xml or build.gradle file.