Module: CSC8104 Enterprise Software and Services

Student: Swapnil Sagar **Student ID:** 250620502

Service Implemented: Hotel Booking Service

---

1. Introduction

The objective of this project was to specify, prototype, validate, and deploy a cloud-enabled distributed enterprise application for a simulated travel agency. The project was based on a modern Java stack, centering around Quarkus, including technologies such as JAX-RS for REST APIs, Java Persistence API (JPA) for data persistence, and Red Hat OpenShift for deployment on cloud infrastructure.

My contribution to this project was in building out the "Hotel" booking service, one of three services in the underlying architecture (Hotel, Taxi, Flight). In this report I will present (Part 1) the architecture of the Hotel service; (Part 2), development of advanced data integrity features; (Part 3), integration of the Hotel service to complete the final distributed TravelAgent application; and a (Part 4) personal reflection of my experience developing the TravelAgent application.

2. Part 1: Base Building Blocks of Service

My primary design principle throughout the project was an emphasis on maintainable and testable architecture for each service. Therefore, I decided to pattern my design using an "Entity-Control-Boundary" (ECB) architecture pattern as illustrated in our course contact example. An ECB architecture provides a very good separation of concerns between facility (in this case the Hotel), the Booking control model and the strings managing the interaction between the controls ("boundaries"), while also separating mounting and service features. Each service is made up of:

- Entity: The JPA data model (for example, Customer.java). This class managed all data-level validation through Bean Validation annotations, such as @Pattern (to enforce rules around "alphabetical string") and @Size (to enforce phone numbers to eleven digits).
- Repository (Control): The database layer. This was the only class to inject the EntityManager and was responsible for all database queries, including em.persist() or named queries.
- Service (Control): The main coordinator (for example, CustomerService.java). It injected the Repository. First it would validate the request and then make a commitment to the database.
- REST Service (Boundary): The public JAX-RS endpoint (for example, CustomerRestService.java). This class' only injection is the Service and is responsible for handling HTTP requests, deserializing JSON and handling exceptions from the Service layer and returning the appropriate HTTP status (for example 409 Conflict).

A central util package was a core part of this architecture with the RestServiceExceptionMapper, enabling me to throw custom exceptions from my Service layer and automatically and consistently format the exception into a clean, JSON error response ({"error": "...", "reasons": {...}}) back to the API consumer.

3. Part 2: Advanced Enterprise Features

Part 2 added functionality to the Service that is critical in supporting data integrity.

3.1. JPA Entity Relationships: Pros and Cons

The first change involved converting the entities from simple Long IDs to full JPA relationships.

• Pros: The main pro was the automatic enforcement of data integrity. By adding @ManyToOne relationships to the Booking entity and the matching @OneToMany(cascade = CascadeType.REMOVE) relationships to the Customer and Hotel entities, deleting a customer now automatically deleted all bookings tied to that customer. This completely avoids the chance of "orphan" data existing in the DB and that is a requirement for an enterprise application.

• Cons: This change contibuted immediately as a StackOverflowError. It was for the same reason as earlier - an infinite JSON serialization loop was occurring: the Customer object referenced a List of Bookings, and each Booking pointed back to the Customer. The fix was to add a @JsonManagedReference is paired with @JsonBackReference and annotation to the @ManyToOne side of the relationship. This broke the recursive loop and allowed the cascade-delete capability to still work.

3.2. JTA Transactions: Definitions and Benefits

The most important aspect of Part 2 was the GuestBooking endpoint which required explicit transaction control.

• A transaction is critical in this case because of the risk of having inconsistent data. The endpoint needs to create a new Customer and their first Booking in a single 'operation', with no way of rollback. If it did not use an explicit transaction there would be no certainty that the data was going to be all or nothing; meaning there is a high likelihood that Customers could be created with no associated Booking.

• JTA Implementation: I added the UserTransaction API as needed. The createGuestBooking method executes userTransaction.begin(), then calls both customerService.createCustomer() and bookingService.createBooking() methods within a try block. Finally, if both the customer and booking were created successfully, userTransaction.commit() is executed. If any exception is caught (e.g., ConstraintViolationException), the catch block calls userTransaction.rollback(), which guarantees the new Customer is discarded from the database.

• Competitive Advantage of JTA: The JTA is an advanced, resource-neutral API. The JTA was used to account for a single database in this project. However, its competitive advantage emerges within larger systems where it can coordinate a single, global transaction across many distinct resources such as two databases, or a database and message queue.

4. Part 3: Integrating into a Distributed System

Part 3 took the application from a single service to a distributed system through a TravelAgent aggregator.

• Architecture: I created a new TravelAgentRestService that now serves as a "facade". Quarkus's @RegisterRestClient is employed; I created three client interfaces

(HotelServiceClient, TaxiServiceClient, FlightServiceClient) to communicate with the commodity services.

• Problems Using my Colleagues' Services:

1. Standardization: The integration of the services was only possible due to the assumption that my colleagues' services were built according to the same API contract (e.g., POST /bookings and DELETE /bookings/{id}). If the developers used a different route, or JSON structure, I would have needed to write custom integration adapter code for each of those scenarios.

2. Availability: If a colleague's service were down, it would cause the entire aggregate booking to fail.

These problems in each integration would be magnified considerably if we had not been discussing, making convenient and shared OpenAPI/swagger documentation the absolute definitive single source of truth.

3. Problems Utilizing my Service: The biggest problem in offering my service was creating a robust service. Any unhandled exception in my code would lead to the failure of all my colleagues' TravelAgent services.

4. Core Challenge (Manual Rollback): The primary challenge was maintaining data consistency without using a distributed transaction. I had to create a manual "compensation" logic to work around this. The TravelAgentService will try to book the Hotel, then the Taxi, then the Flight. If any of these calls fail (e.g., the Flight service returns a 409 Conflict), the catch block is executed. Then the catch block calls the endpoint DELETE /bookings/{id} for any of the services that succeeded (e.g., cancelling the Hotel and Taxi bookings), thus manually rolling back the operation.

5. Reflection on DevOps and Automated Testing

Automated testing was more than just a requirement; it was essential for me to succeed. Writing a test for every feature was the only way to find and correct the complex bugs I had encountered along the way. The testTransactionRollbackOnFailure test, for example, was the only way to prove that the JTA rollback logic was working properly. This test simulates sending an invalid booking, verifies that we get a 400 response, and immediately sends a second call to GET /customers to assert that the customer was never written. Having a test-driven mindset and the CI/CD workflow that we pushed to GitHub to trigger the build on OpenShift in the deployment process made me feel confident in deploying my transactional services knowing that they worked properly.

6. Personal Reflection

This project was a challenging but ultimately very rewarding learning experience. The "easy" part was employing the ECB pattern to support building the base service develop a cabinet. The "hard" part was not really about the features themselves, but rather debugging the complex, non-obvious interactions between the enterprise components in the system.

I also encountered many hard bugs that made me really dive into the technology, including:

1. Validation Mismatch - my first tests failed with the 400 Bad Request because my test data ("Swapnil Sagar") did not match my validation rule (^[A-Za-z]+$). This was a good lesson that what I had specified was not the same as what I had coded (@Pattern), and my test data (import.sql) must all be perfectly in sync.

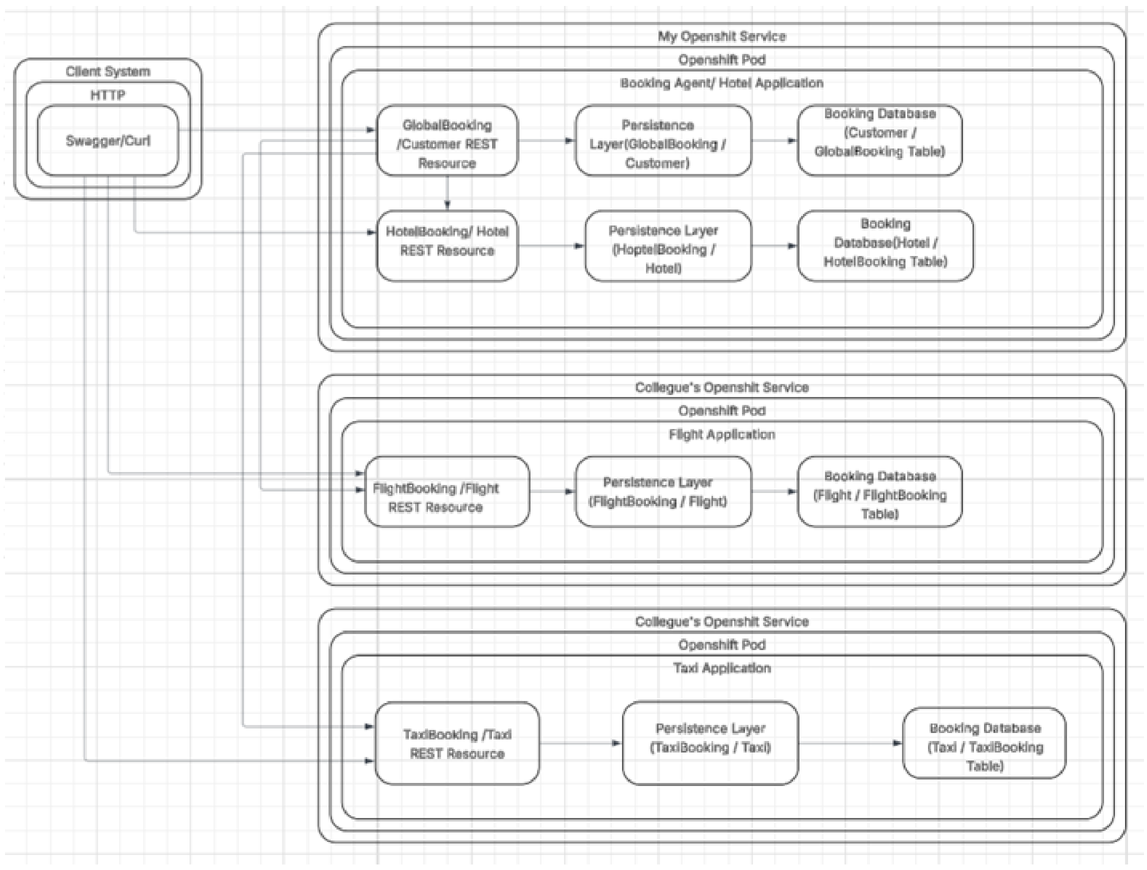
2. Primary Key Collision: My Booking tests resulted in failure due to a 500 error. The log indicated a JdbcSQLIntegrityConstraintViolationException. It turned out that my import.sql file created a booking with id=1, but the GenerationType.TABLE generator started at 1 as well.

3. Deployment Failure: My app crashed on OpenShift with Connection refused. I discovered that this was because my REST Clients were set to localhost:8080, and the app was trying to connect to itself before starting completely. To correct this, I used Quarkus's test profiles: making the test URLs equal to ${quarkus.http.port} while the production URLs were equal to the live OpenShift URLs.

4. The Hardest Bug: The last bug was an org.hibernate.PersistentObjectException: detached entity passed to persist. This occurred in Part 3 as I was saving the TravelAgentBooking. The Booking object that got returned from my HotelServiceClient was simply a "detached" POJO. The solution was to take that ID and do an em.find() to fetch the managed version of the entity first before saving.

Overall, this project was a demonstration of how enterprise frameworks, such as JPA and JTA, solve many important and difficult problems, but the process usually adds new layers of complexity to the application, such as entity lifecycles, distributed state, which can only be managed with a robust, test-driven development process.

Architecture: -


Client System
HTTP
Swagger/Curl
My Openshit Service
Openshift Pod
Booking Agent/ Hotel Application
GlobalBooking /Customer REST Resource
Persistence Layer(GlobalBooking / Customer)
Booking Database (Customer / GlobalBooking Table)
HotelBooking/ Hotel REST Resource
Persistence Layer (HoptelBooking / Hotel)
Booking Database(Hotel / HotelBooking Table)
Collegue's Openshit Service
Openshift Pod
Flight Application
FlightBooking /Flight REST Resource
Persistence Layer (FlightBooking / Flight)
Booking Database (Flight / FlightBooking Table)
Collegue's Openshit Service
Openshift Pod
Taxi Application
TaxiBooking /Taxi REST Resource
Persistence Layer (TaxiBooking / Taxi)
Booking Database (Taxi / TaxiBooking Table)