Enterprise Middleware Report

Name : Prabhu Yogesh Vijayan
ID : 2206116113

Part 1

The 3 basic Rest Services in the application are Customer, Taxi and Bookings. Each have a multitude of endpoints that include creation, deletion and get methods. All of the 3 have a rather similar structure so looking at one(Customer) will give overall rough overview of the code flow and structure

To start of the customer JAX-RS rest service has a CustomerRestService controller class which acts the main controller where the endpoints hits. It consumer the type JSON and return the type JSON as well. The paths for the several methods are defined here using the @Path annotation. The methods also define the API responses for each error codes to be shown in Swagger. Methods that require persisting, updating or deleting are annotated with @Transactional.

The Customer Service is injected into the CustomerRestService using the @Inject annotation. This class acts as a medium between the Rest controller and the CustomerRepository class. CustomerService is annotated with @Dependent annotation. The service layer is also where any data manipulation or validation is done. The CustomerValidator class is then called where the request is then validated against the predefined conditions as well as any data in the database for conflict errors if any. The methods here then subsequently call the repository class when the transactions happens.

The CustomerRepository class which injected into the CustomerService is then called from there. Here the EntityManager is then injected into the class. Here the entitymanager then persists, retrieves or updates any data required. Name queries are used here which are defined in the respective entity classes of the Customer. This makes it faster as the queries are cached during compile time and the queries are run when invoked. The data passed through the API is then persisted in the database.

There are 3 different entities define in the first part of the project. The Customer and Taxi entity are separate with the required fields. However, both of them are individually connected to the Booking entity in the second part by the below relations.

```
@JsonIgnore
@OneToMany(mappedBy = "customer", cascade = CascadeType.REMOVE)
private List<Booking> booking;
```

The @JsonIgnore annotation ensures that there is no issue of recursive definition. The mapping means that one customer can have many or a list of bookings. Furthermore the cascade of remove ensures that when the customer is deleted the associated booking is also deleted at the same time. This is the same case as for taxi.

Part 2

In this part the Booking entity is linked to the Taxi and Customer entity via Many to One relation. This means that one Customer or Taxi can have multiple bookings. Hence the @ManyToOne annotation was used in the Booking entity and @OneToMany annotation was used at the Customer/Taxi Entities.

The benefit of this use of foreign key is that it supports cascading deletion which allows entries that are related to be deleted in one go. There are also several cascade types like MERGE, PERSIST and DELETE.

Moreover JPA lets you work with a higher level of abstraction and separates the SQL from business logic. It also lets the code to be accessed by any wide range of Databases. Also JPA framework uses only unchecked exceptions hence, we don't need to catch or declare them at every place we're using them.

However there was several issues I faced while using this mapping. Sometime a foreign key constraints issue when persisting or deleting. And the part of figuring out the bi directional mapping for the entity relationship was confusing and required several trials and errors to get it right. JPA allows us to code Java applications using object-oriented principles and best practices without having to worry about database semantics.

For transaction in the GuestBooking part instead of using @Transactional the legacy JTA approach was implemented. This is because in this there are 2 distinct transactional services being called and might result in issues. Also furthermore if the transaction should fail both of them should be rolled back.

For example if customer is created but Booking has issues both should be rolled back. If automated transaction was used the customer would be saved but booking would not be persisted. However with JTA we are able to commit the transaction only at the end after both have been persisted. Any exceptions caused would cause the entire transaction to roll back and hence not get saved.

In GuestBooking the entire approach is different from the previous in that it uses a Data Transfer Object instead of an entity. Hence since the customer and booking are together in the JSON they are represented just as an DTO. First the customer is created and then the customer id is set in the booking which then calls the booking service.

Part 3

One of the issues of using different services by my colleagues was that there were times when their services was unavailable. Hence when I was trying to make a travel agent booking their service was failing and resulting in delays in testing my code. However, since they were all in the same class, I could communicate with them to bring the service up and continue testing my cases. However, if they had not been in the same room then I would have had to raise a ticket and wait a significant amount time before the service was up and running again.

Furthermore, each of the other services had created a different format in which they were had defined customers. So, it would have required to create a different customer DTO for each of the service. However, since they were all in the same it was possible to coordinate and ensure that the same customer format was used throughout the project so that project flow would be the same.

Working of Travel Agent

@NotNull
@OneToOne(cascade = { *CascadeType.**REMOVE*** })
@JoinColumn(name = "booking_id")
private Booking taxiBooking;

The one to one bidirectional mapping ensures that when the travel agent booking is deleted the corresponding booking is also deleted in the booking tables.

Catch blocks in the creation of travel agent booking ensures that if there is a failure in any of the other services the data is not persisted in the other DBs and is deleted.

```
} catch (Exception e) {

    /*
     * delete method for hotel and flight to roll back
     */

    hotelService.deleteHotelBooking(hotelBooking.getId());
    foreignTaxiService.deleteTaxiBooking(foreignBooking.getId());
```