Design Documentation

1 Functionalities

1.1 For public users

1. Search hotels with various options such as hotel attributes, room attributes, availability, aggregated numbers.
2. Room recommendations. Optimize numbers of adults, children, beds, and rooms using linear programming.
3. Map to show hotel locations.

1.2 For app users

1. User registration and authentication.
2. Reserve rooms with optional no prepayment days and free cancellation days.
3. Payment with Paypal.
4. Save favorite hotels.
5. Booking records by status and dates.

1.3 For hotel managers

1. Hotel and rooms registrations and modification with address recocommendations and image uploads
2. Hotel summary page for hotel information, reservation, and booking history
3. Manage available and reservation dates for each room through a timeline calendar
4. Booking records by status and dates

2 Non-functionalities

1. Scalability
   (a) All microservices are horizontally scalable.
   (b) The MySQL database used for availability and booking is expected to have highest loads. This can be handled by vertically sharding database based on hotel IDs.

   (c) Cassandra, Elasticsearch, and Kafka are horizontally scalable.

2. Low latency

   (a) Redis is used for the Booking MySQL database to retrieve recently used booking information.
   (b) OAuth 2.0 with JWT has an advantage of identifying users without querying any database once authenticated by the OAuth resource server.

3. Low cost

   (a) Only modified room or price information are sent from the Booking MySQL to an Elasticsearch cluster to make Kafka light-weighted.
   (b) Cancelled or completed records of booking or reservation will be archived from a MySQL database to a Cassandra cluster, enabling a lower cost search in the MySQL database.

4. Security

   OAuth 2.0 with JWT token is used to authenticate users and retrieve user or hotel manager ids which are used to verify access to data resource.

5. Strong consistency

   (a) User, hotel, dates, and booking information are modified using the row lock functionality of MySQL with timeout. MySQL clusters use two-phase commits for strong consistency between nodes.
   (b) Write operations to Elasticsearch is controlled by version control and its optimistic concurrency control mechanism for data consistency. Also, as a response to the intrinsic inconsistency problem of Elasticsearch, the data related to booking will be overwritten whenever booking request validation has failed. Finally, Elasticsearch is eventually consistent for reading data. However, the updates occurs near real-time given that search consumer services are not overloaded.
   (c) While Cassandra is eventually consistent, since only archived data will be stored, Cassandra will show strong consistency in this system.
   (d) Kafka is highly durable by topic replication and leader selection, allowing high consistency between data stored in Hotel MySQL, Booking MySQL, and Elasticsearch.

6. Fault tolerance

   Each microservice uses threads for each request and handles errors made while processing the request unless there is an error in connection with databases, which can have high availability due to horizontal scaling.

7. High availability

   Horizontal scaling and Fault tolerance enables high availability.

8. Durability

(a) A MySQL cluster has durability by maintaining a transaction log file.

(b) Cassandra saves information with duplicates.

(c) Elasticsearch is not used as a primary data source due to its low durability.

3 System Design

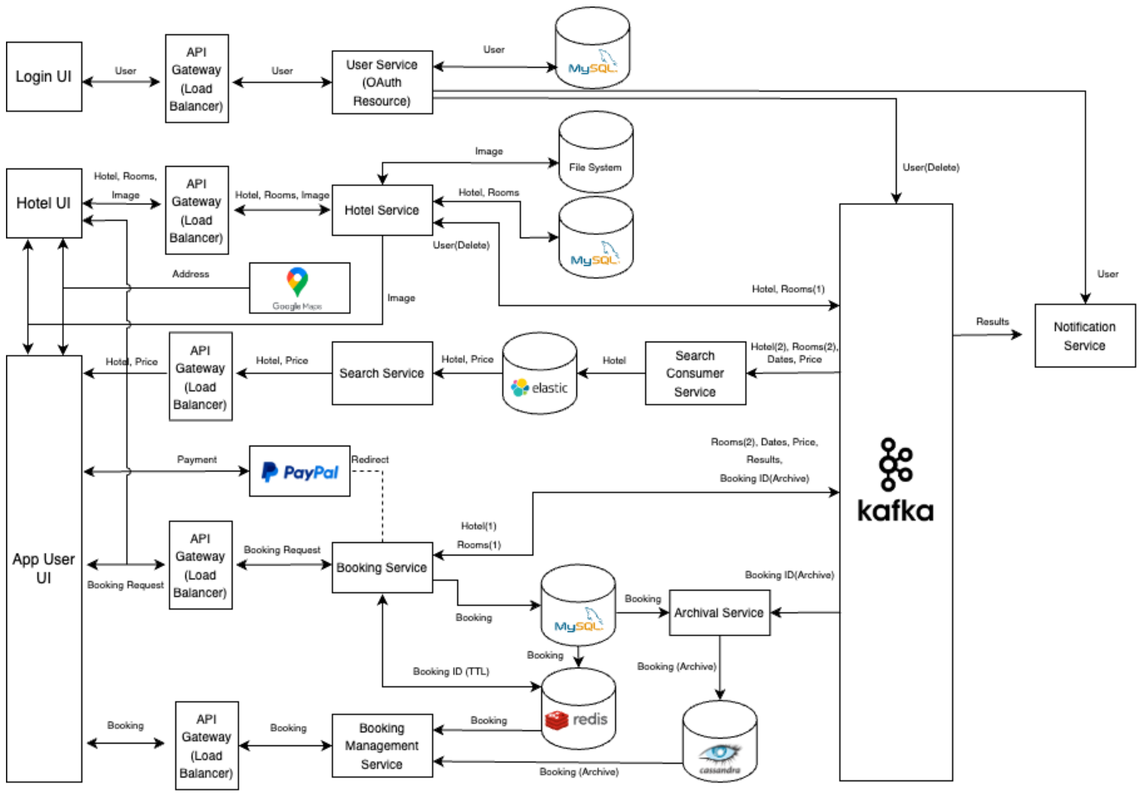


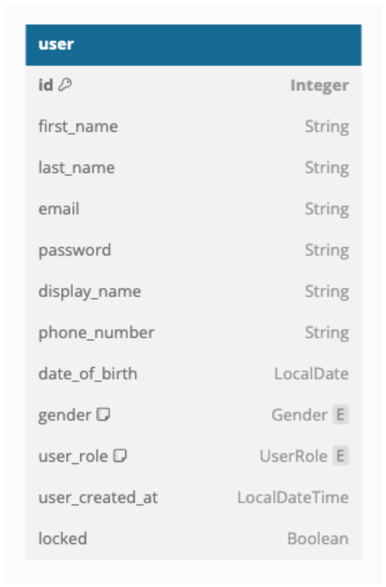
Figure 1: System design schema with abstract data flow

3.1 User service

User registration and information modification are processed by this service. This service uses a MySQL database to save data. This service also works as a OAuth 2.0 resource server that generates JWT tokens for users. Any registered app user can also be a hotel manager by adding their own hotel.

Hotel images can be uploaded by frontend and they will be transformed, compressed, and saved in a file system. Due to large storage capacity, images are currently not shown in the hotel view pages in the demo.

When a user is deleted, all hotels registered by this hotel are also deleted together.



user	
id	Integer
first_name	String
last_name	String
email	String
password	String
display_name	String
phone_number	String
date_of_birth	LocalDate
gender	Gender E
user_role	UserRole E
user_created_at	LocalDateTime
locked	Boolean

Figure 2: Database schema of User MySQL

3.2 Hotel service

This service is used for hotel managers to save and modify hotel and rooms information. The Hotel MySQL database is used to save the information and works as a primary data source. Each hotel entity can have multiple rooms entity which represents multiple actual rooms with same properties. Rooms can be activated with quantity and available date ranges while detailed availabilities can be updated through the booking service.

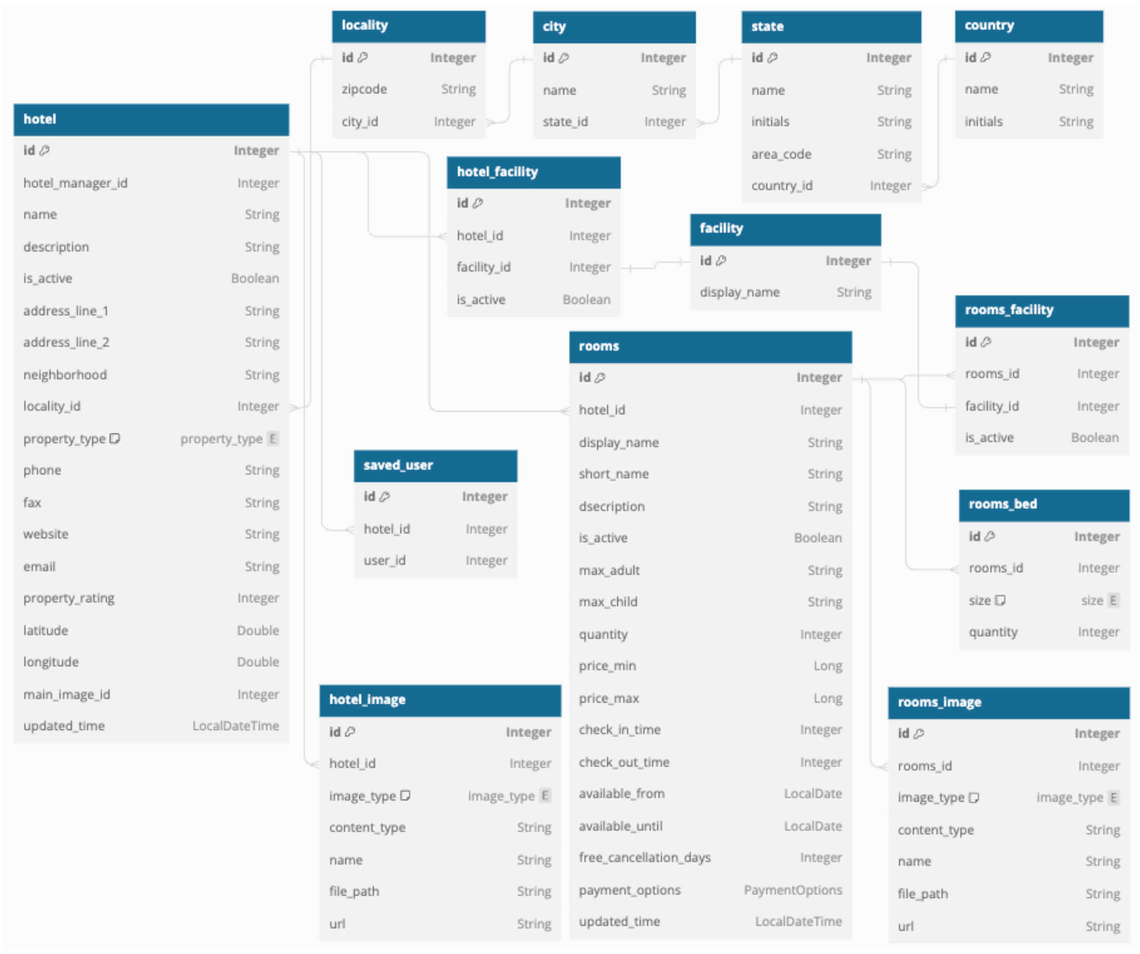


Figure 3: Database schema of Hotel MySQL

3.3 Booking service

The booking service manages rooms availability and processes reservations.

1. Dates for each room and price for each rooms entity for each date are generated based on the activeness, quantity, available date range, minimum price, and maximum price provided by the hotel service. These information will be propagated to the Elasticsearch component

2. For rooms reservation requests, this service first verifies availability, price, check-in time, check-out time, no prepayment day, and free cancellation day. Then, along with the guest information a new reservation record is created and saved in the database.

3. For payment, the booking ID is sent to a Redis cluster with a timeout of 5 minutes and the user is redirected to PayPal to enter payment information. Once the payment information is validated by PayPal, the user is redirected to the Booking service which requests an execution to PayPal. However, if the booking ID is not found from the Redis cluster,

the booking status will be changed to cancelled due to timeout and the user will be redirected to the hotel page. If the payment is executed successfully, the user will again be redirected to a frontend page.

4. A cron job is executed once each day to remove past date availability, add a new available date, and add a price for the new date. This will be done up to 100 hotels at a time.

5. A cron job is executed to cancel all room reservations where the payment is not received until the deadline.

6. Booking information can be updated by both hotel and user unless their status has changed to terminal state which are either cancelled or completed. All updates are made after verifying the owner of the data either based on the hotel manager's user ID or the app user's user ID. Hotel managers can view and update availability, booking dates, and booking rooms using a calendar timeline in the frontend.

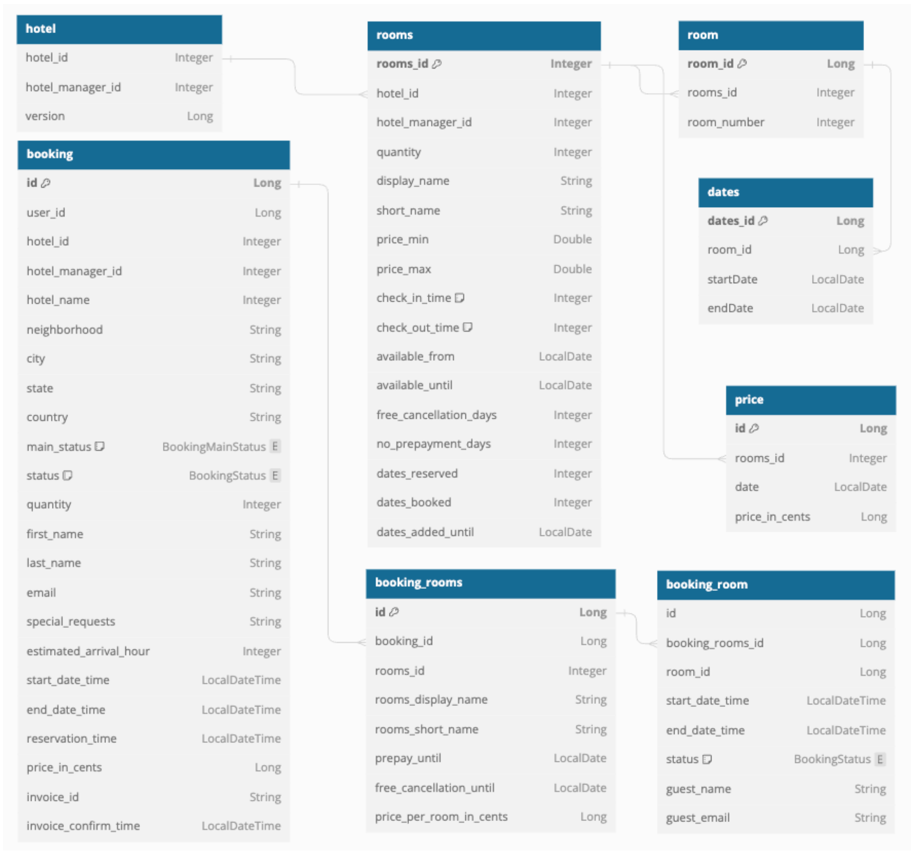
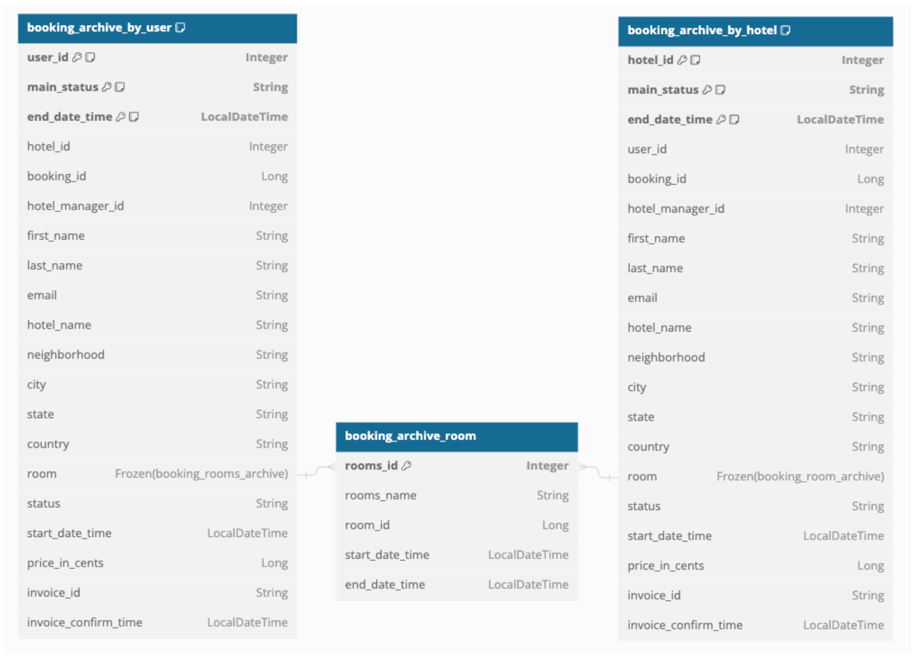


Figure 4: Database schema of Booking MySQL

3.4 Archival service

The archival service moves data from Booking MySQL to a Cassandra cluster to archive the data. For cancelled reservations, the archiving is requested by the Booking service through

Kafka. A cron job is executed daily to archive completed or any outdated data.



booking_archive_by_user	
user_id	Integer
main_status	String
end_date_time	LocalDateTime
hotel_id	Integer
booking_id	Long
hotel_manager_id	Integer
first_name	String
last_name	String
email	String
hotel_name	String
neighborhood	String
city	String
state	String
country	String
room	Frozen(booking_rooms_archive)
status	String
start_date_time	LocalDateTime
price_in_cents	Long
invoice_id	String
invoice_confirm_time	LocalDateTime

booking_archive_room	
rooms_id	Integer
rooms_name	String
room_id	Long
start_date_time	LocalDateTime
end_date_time	LocalDateTime

booking_archive_by_hotel	
hotel_id	Integer
main_status	String
end_date_time	LocalDateTime
user_id	Integer
booking_id	Long
hotel_manager_id	Integer
first_name	String
last_name	String
email	String
hotel_name	String
neighborhood	String
city	String
state	String
country	String
room	Frozen(booking_room_archive)
status	String
start_date_time	LocalDateTime
price_in_cents	Long
invoice_id	String
invoice_confirm_time	LocalDateTime

Figure 5: Database schema of the Cassandra cluster

3.5 Booking management service

This service is used to retrieve booking data for users for both active and archived reservation data. It also aggregates the results for the hotel manager.

3.6 Search consumer service

The search consumer service listens to Kafka messages sent by Hotel service or Booking service and updates hotel information to Elasticsearch as an alternate to Logstash. Only relevant data from both services are aggregated and saved. Sequence number and primary term are used as the optimistic concurrency control mechanism to ensure that there is no concurrent updates for the same document.

While hotel information is created or updated directly from hotel service through Kafka, rooms information is created or updated by booking service. All entity IDs match with the original IDs in the MySQL databases and only the modified data are sent through Kafka for dates and price updates.

3.7 Search service

The search service is used to generate and execute queries to the Elasticsearch component. The main hotel search query performs the following actions.

1. Filter hotel by location, property type, property rating, and hotel facilities.
2. Nest to rooms and filter by rooms facilities.
3. Nest to room and dates and filter by available date range.
4. Aggregate by hotel ID and get the sum of maximum adults, maximum number of people, number of beds, and number of rooms.
5. Filter hotels based on the number of maximum adults, maximum number of people, number of beds, and number of rooms.
6. Append relevant columns of hotel hits to the result.
7. Aggregate price for each rooms over dates.
8. Append relevant columns of available rooms with their available quantity and aggregated price to the results.

Once the search service receives and parses the query results, it runs linear programming to optimize room recommendation based on the number of adults, number of people, number of beds, number of rooms while matching the price range criteria. The linear programming runs only a number of iterations linear to the number of available rooms for low latency. A small randomness is introduced to the result to allow diverse results for the same query.

The second search query of the search service is the price aggregation which is later validated by the booking service for new reservations or bookings.

Finally, search service provides number of hotels based on city and property types for the main page of the frontend.

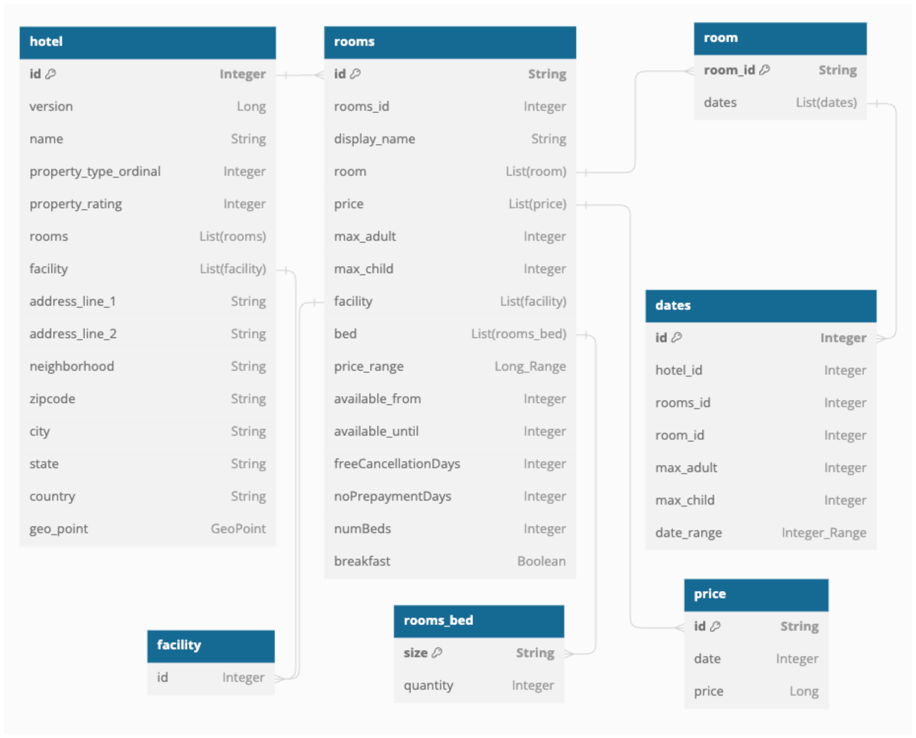


Figure 6: Database schema of the ElasticSearch cluster

3.8 Notification Service

The notification service listens events from Kafka, fetches user information from the User service, and sends email to hotel manager or users. The email functionality is currently disabled.

3.9 Supplementary services

3.9.1 API Gateway

A Spring Cloud API Gateway is used to route and load-balance requests. In deployment with Kubernetes, the routing is replaced by Ingress and load-balancing is replaced by Service of each microservices.

3.9.2 Zipkin

Zipkin is a distributed tracing system to track used to gather time consumption for processing each request. This service is not used in the Kubernetes deployment as well.

3.9.3 Eureka-server

This service provided in the Spring Framework is used to discover services for the API Gateway and Zipkin services to activate.

3.9.4 Kibana

Kibana is a frontend of Elasticsearch providing search and data visualization.

4 Frontend

5 To build run this project

5.1 Configurations

1. Set base image url address (custom.file.imageUrlPrefix) in "hotel/src/main/resources/application-${profile}.yaml".
2. Set backend ingress address (custom.paypal.host) and frontend address (custom.paypal.frontend) in "booking/src/main/resources/application-${profile}.yaml" for redirections.

5.2 Run manually

1. Modify database urls for each service in "${service-name}/src/main/resources/application-default.yaml".
2. Compile java files.

   ```
   mvn compile
   ```

3. Run all databases and Zipkin.

   an example using docker:

   ```
   docker compose up -f docker-compose-db.yaml -d
   ```

4. Run all services.

   ```
   cd \$\{service-name\}

   SPRING\_PROFILES\_ACTIVE=default java -jar target/\$\{service-name\}-0.0.1-SNAPSHOT.jar
   ```

5.3 Run with Docker

1. Compile java files.

   ```
   mvn compile
   ```

2. Compile Docker images with new image names in "docker-compose.yml".

   ```
   docker compose build
   ```

3. Run Docker files.

   ```
   docker compose up -d
   ```

4. Wait until a superuser for Cassandra is created (a few minutes).

   Cassandra logs:

   ```
   docker logs -f hb-cassandra
   ```

5. Create Cassandra keyspace and tables.

   ```
   sh scripts/init.sh
   ```

5.4 Run with Kubernetes (Minikube)

1. Compile java files.

   ```
   mvn compile
   ```

2. Compile Docker images with new image names in "docker-compose.yml" and push them to a Docker .repository.

   ```
   docker compose build
   docker push ${repository-name}/${image}
   ```

3. Change "k8s/minikube/services" to used the new images.

4. Start Minikube.

   ```
   minikube start
   ```

5. Create and run all databases: Cassandra, ElasticSearch, Kafka, MySQL(x3), Redis.

```
cd k8s/minikube

# Setting default username and password for ElasticSearch
kubectl create secret generic hb-elasticsearch-es-elastic-user \
    --from-literal=elastic=changeme

kubectl apply -f bootstrap/cassandra
kubectl apply -f bootstrap/kafka
kubectl apply -f bootstrap/mysql-booking
kubectl apply -f bootstrap/mysql-hotel
kubectl apply -f bootstrap/mysql-user
kubectl apply -f bootstrap/redis
kubectl apply -f bootstrap/elastic-search/crds.yaml
kubectl apply -f bootstrap/elastic-search/operator.yaml
kubectl apply -f bootstrap/elastic-search/elasticsearch-config.yaml
kubectl apply -f bootstrap/elastic-search/kibana-config.yaml
```

6. Manually create Cassandra keyspace and table once superuser is created.

   ```
   kubectl logs -f hb-cassandra-0 # to check logs
   sh bootstrap/cassandra/init.sh
   ```

7. Run all services.

   ```
   kubectl apply -f services/user
   kubectl apply -f services/hotel
   kubectl apply -f services/search-consumer
   kubectl apply -f services/search
   kubectl apply -f services/booking
   kubectl apply -f services/booking-management
   kubectl apply -f services/archival
   kubectl apply -f services/notification
   ```

8. Run ingress with Kong ingress controller and test connection.

   ```
   kubectl apply -f ingress
   minikube addons enable kong
   export PROXY_IP=$(minikube service -n kong kong-proxy --url | head -1)
   echo $PROXY_IP
   curl -i $PROXY_IP/api/v1/user/test
   ```

9. Port forward.

   ```
   kubectl port-forward -n kong --address 0.0.0.0 service/kong-proxy 8001:80
   ```