Project Explanation

So, in this project, we were already been given 4 Micro-services

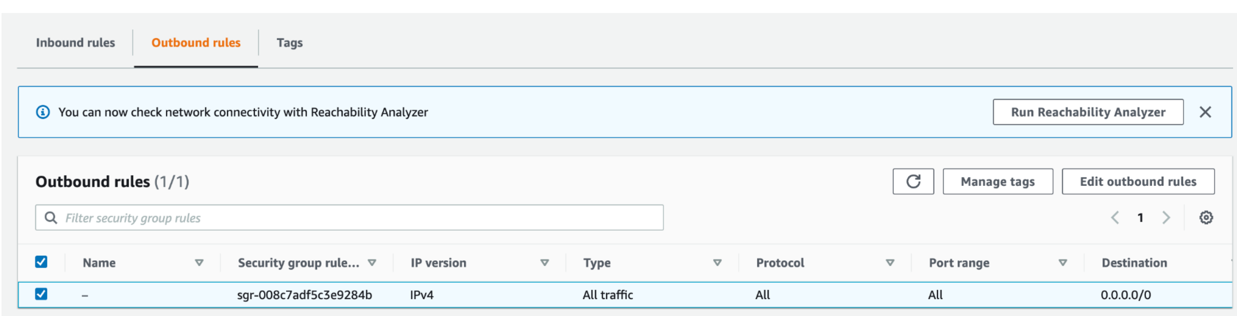
1. Booking Service: `booking-service`
2. Payment Service: `payment-service`
3. Notification Service: `notification-service`
4. Eureka Server: `eureka-server`

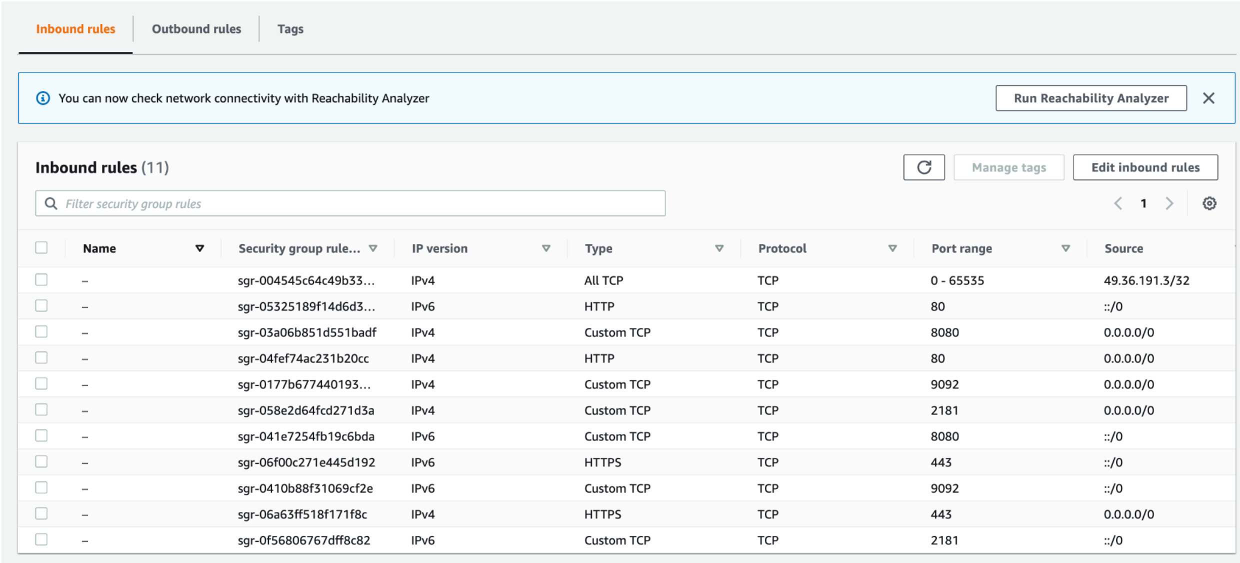
We needed to compose and deploy the whole project. For the same, here the main steps that we took:

- We updated `application.properties` => `spring.datasource.url` for Booking Service and Payment Service respectively with the Amazon RDS instance URL that we got from the creation of the Amazon RDS instance.
- We updated the IP address of the Kafka Server (EC2 Instance) that we got from the creation of the Amazon EC2 instance for the Kafka Server: Booking Service (Producer) and Notification Service (Consumer) respectively.
- We created `Dockerfile` for each of the 4 micro-service: Eureka Server, Booking Service, Payment Service and Notification Service respectively.
- We created a single `docker-compose.yml` file in the root of the project to compose the deployment of the overall project in one go to create a Multi Container Deployment.

Now, Let us document the overall step by step process of the whole deployment process.

1. First, we created a VPC and Security group respectively referring from this [resource](). Please find the screenshots attached for the Inbound and Outbound rules that we set in our security groups of our VPC. This also includes My IP Address from our local home desktop/server to connect to both of these instances.





Inbound rules | Outbound rules | Tags

You can now check network connectivity with Reachability Analyzer

Run Reachability Analyzer

Inbound rules (11)

Manage tags | Edit inbound rules

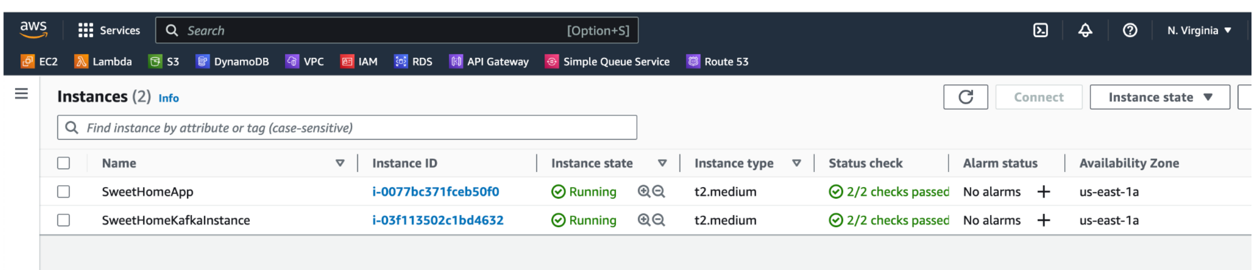
Filter security group rules

Name	Security group rule...	IP version	Type	Protocol	Port range	Source
–	sgr-004545c64c49b33...	IPv4	All TCP	TCP	0 - 65535	49.36.191.3/32
–	sgr-05325189f14d6d3...	IPv6	HTTP	TCP	80	::/0
–	sgr-03a06b851d551badf	IPv4	Custom TCP	TCP	8080	0.0.0.0/0
–	sgr-04fef74ac231b20cc	IPv4	HTTP	TCP	80	0.0.0.0/0
–	sgr-0177b677440193...	IPv4	Custom TCP	TCP	9092	0.0.0.0/0
–	sgr-058e2d64fcd271d3a	IPv4	Custom TCP	TCP	2181	0.0.0.0/0
–	sgr-041e7254fb19c6bda	IPv6	Custom TCP	TCP	8080	::/0
–	sgr-06f00c271e445d192	IPv6	HTTPS	TCP	443	::/0
–	sgr-0410b88f31069cf2e	IPv6	Custom TCP	TCP	9092	::/0
–	sgr-06a63ff518f171f8c	IPv4	HTTPS	TCP	443	0.0.0.0/0
–	sgr-0f56806767dff8c82	IPv6	Custom TCP	TCP	2181	::/0

2. Then we launched a couple of EC2 instances in our VPC referring from this [resource](). Both were created as a t2.medium EC2 instance.
    a. First EC2 instance is for running the **Main server** and we choose Ubuntu operating system for the same server.
    b. Second EC2 instance is for running the **Kafka server** and we choose the Amazon AMI operating system for the same server.

Please note that if you check the above screenshot in step 1, you will find that we have set the necessary ports in both **Main Server and Kafka Server** to access things already**.**

Also, Please find the attached screenshot for the running servers.



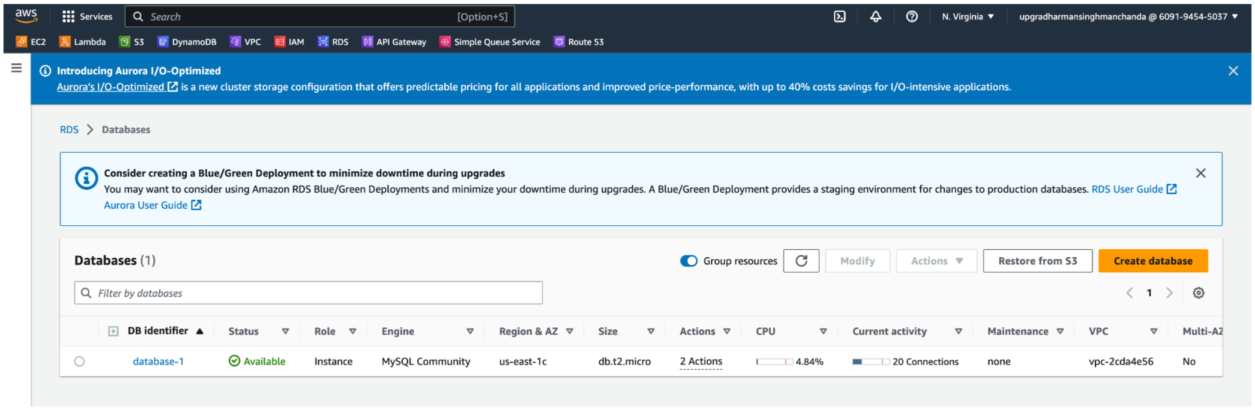
Instances (2) Info

Find instance by attribute or tag (case-sensitive)

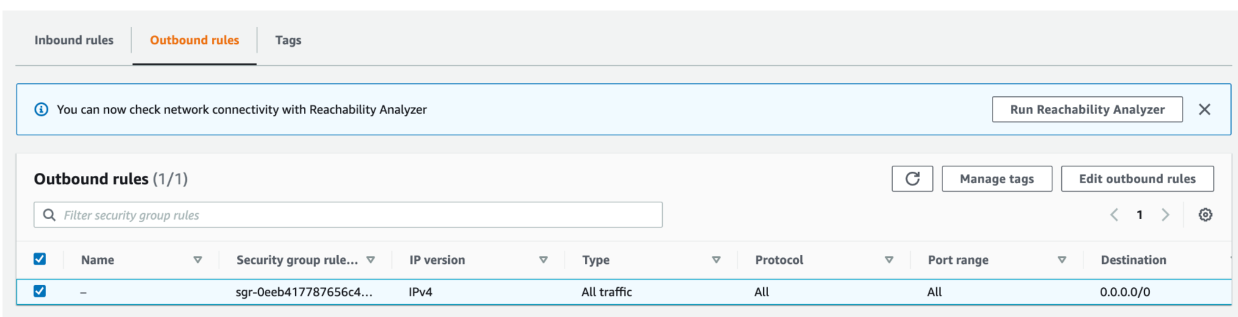
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
SweetHomeApp	i-0077bc371fceb50f0	Running	t2.medium	2/2 checks passed	No alarms	us-east-1a
SweetHomeKafkaInstance	i-03f113502c1bd4632	Running	t2.medium	2/2 checks passed	No alarms	us-east-1a

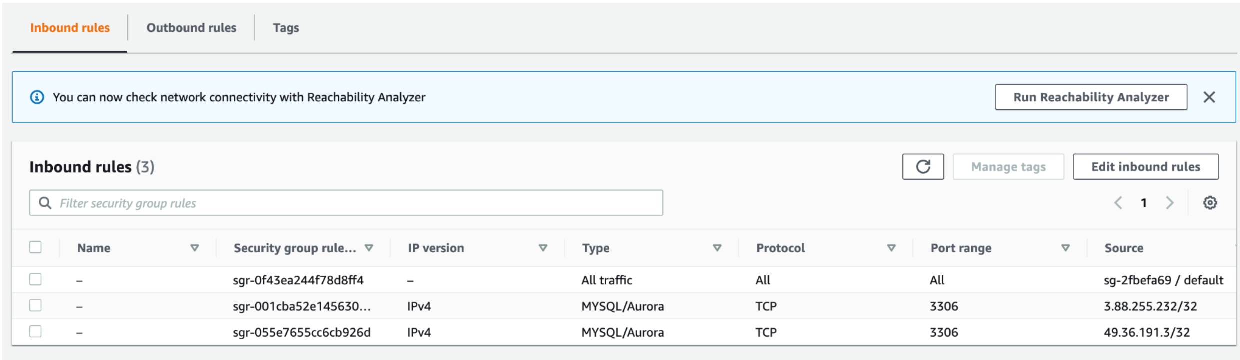
3. We store the AWS key-pairs for the instances in one level up directory of the `Sweet-home` folder. In our case, it's `hotel-booking-monorepo`. We store our key pairs in `hotel-booking-monorepo/aws-keys` (that we won't be sharing as not needed and concerns privacy), whereas the project lives at `hotel-booking-monorepo/Sweet-home`
4. The EC2 instances have been created, and now it is time to log in and gain access to the instance and thus we now log in to our instances, referring from this [resource]().
5. In the **Main server** EC2 instance, we install **Docker** referring from this [resource]().

6. Now, we create a single RDS instance referring from this [resource](). Please find the screenshot for the running Database server.



7. The RDS instance were set with following specifications to match the `application.properties` code that were given
   - User name: admin
   - Password: upgrad123
8. Now for the RDS Instance, we will use the default VPC and Security group that were given by AWS for the region. For this specific security group, please find the screenshots attached for the Inbound and Outbound rules that we set in this one. This not only includes My IP Address from our local home desktop/server to connect to the database instance but also includes the IP Address of the Main Server EC2 instance to connect to the database instance respectively.





9. Now we will create separate databases: SweetHomeBooking and SweetHomePayment respectively for Booking Service and Payment Service to host the booking and payment table based on the Sweet-Home schema.
10. Now we update `application.properties` => `spring.datasource.url` for Booking Service and Payment Service respectively with the Amazon RDS instance URL that we got from the creation of the Amazon RDS instance.
11. In the **Kafka server** EC2 instance, we now set the elastic IP to ensure that the IP does not change every time the EC2 instance is logged into. For the same, we refer to this resource.
12. Now in the **Kafka server** EC2 instance, we install Kafka referring from this resource.
13. Now we update the IP address of the Kafka Server (EC2 Instance) that we got from the creation of the Amazon EC2 instance for the Kafka Server: Booking Service (Producer) and Notification Service (Consumer) respectively.
14. Now it's time to create `Dockerfile` for each of the 4 micro-service: Eureka Server, Booking Service, Payment Service and Notification Service respectively. Before creating dockerfiles, we looked at Postman API Documentation and the already coded given `application.properties`. We came to the conclusion that we will use and expose these below ports for our Services within docker:
    a. Booking Service: PORT `8080`
    b. Payment Service: PORT `8083`
    c. Notification Service: Not Applicable (N/A)
    d. Eureka Server: PORT `8761`
15. To make sure that we don't have to create the JAR files manually, we use a multistage docker file approach so that `docker compose` (that we install soon) will automatically create the jar file of the application and then create a lightweight image of it respectively.

16. For Booking Service, we use this below multistage docker file,

```
# Stage 1: Build the application
FROM maven:3.8.1-jdk-11 AS build
WORKDIR /usr/src/app
COPY src ./src
```

```
COPY pom.xml .
RUN mvn clean install -DskipTests

# Stage 2: Create the lightweight image
FROM gcr.io/distroless/java
COPY --from=build /usr/src/app/target/bookingService.jar
/usr/src/app/bookingService.jar
WORKDIR /usr/src/app
ENV PATH="${PATH}:${JAVA_HOME}/bin"
EXPOSE 8080
ENTRYPOINT [ "java", "-jar", "/usr/src/app/bookingService.jar"]
```

17. For Payment Service, we use this below multistage docker file,

```
# Stage 1: Build the application
FROM maven:3.8.1-jdk-11 AS build
WORKDIR /usr/src/app
COPY src ./src
COPY pom.xml .
RUN mvn clean install -DskipTests

# Stage 2: Create the lightweight image
FROM gcr.io/distroless/java
COPY --from=build /usr/src/app/target/paymentService.jar
/usr/src/app/paymentService.jar
WORKDIR /usr/src/app
ENV PATH="${PATH}:${JAVA_HOME}/bin"
EXPOSE 8083
ENTRYPOINT [ "java", "-jar", "/usr/src/app/paymentService.jar"]
```

18. For Notification Service, we use this below multistage docker file,

```
# Stage 1: Build the application
FROM maven:3.8.1-jdk-11 AS build
WORKDIR /usr/src/app
COPY src ./src
COPY pom.xml .
```

```
RUN mvn clean install -DskipTests

# Stage 2: Create the lightweight image
FROM gcr.io/distroless/java
COPY --from=build
/usr/src/app/target/notificationService-jar-with-dependencies.jar
/usr/src/app/notificationService-jar-with-dependencies.jar
WORKDIR /usr/src/app
ENV PATH="${PATH}:${JAVA_HOME}/bin"
ENTRYPOINT [ "java", "-jar",
"/usr/src/app/notificationService-jar-with-dependencies.jar"]
```

19. For Eureka Server, we use this below multistage docker file,

```
# Stage 1: Build the application
FROM maven:3.8.1-jdk-11 AS build
WORKDIR /usr/src/app
COPY src ./src
COPY pom.xml .
RUN mvn clean install -DskipTests

# Stage 2: Create the lightweight image
FROM gcr.io/distroless/java
COPY --from=build /usr/src/app/target/eurekaServer.jar
/usr/src/app/eurekaServer.jar
WORKDIR /usr/src/app
ENV PATH="${PATH}:${JAVA_HOME}/bin"
EXPOSE 8761
ENTRYPOINT [ "java", "-jar", "/usr/src/app/eurekaServer.jar"]
```

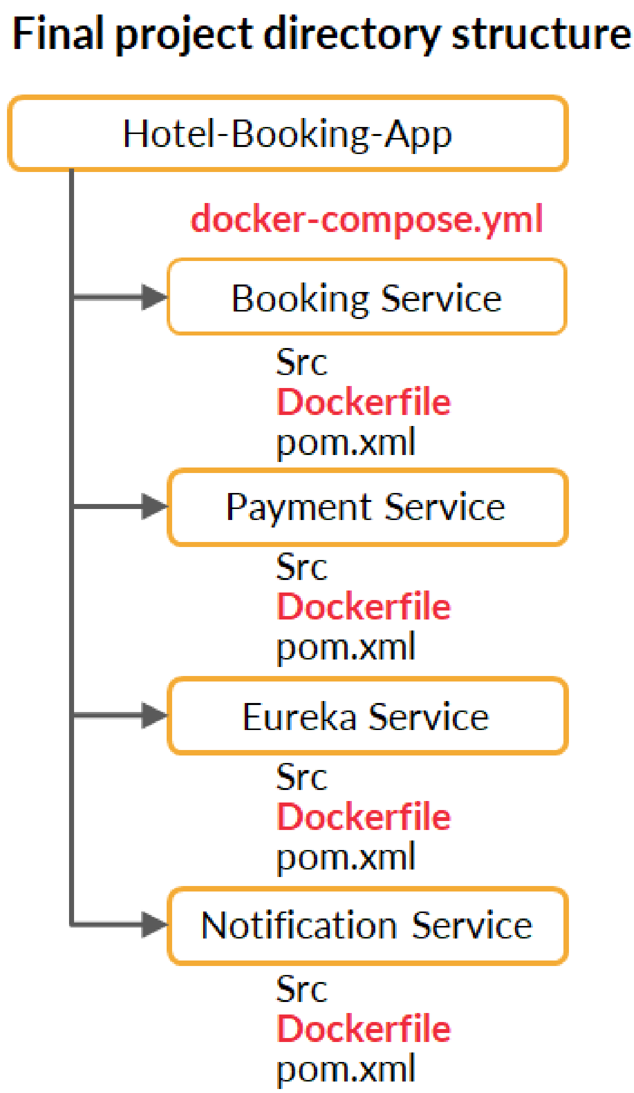
20. Overall, all these Dockerfile(s) in these 4 microservices builds the application using Maven in one stage and then creates a lightweight image using a minimal Java runtime environment in another stage. The final image(s) runs the application by executing the JAR file within the container.
21. Now to create a Multi Container Deployment, we install **Docker compose** referring from this resource.
22. Now we create a single `docker-compose.yml` file in the root of the project to compose the deployment of the overall project in one go to create a Multi Container Deployment. Here is the code of `docker-compose.yml` file:

```yaml
version: '3.3'

services:
  bookingsvc:
    build: bookingservice
    container_name: bookingsvc
    image: hotelbooking/bookingsvc:1.0.0
    ports:
    - "8080:8080"
    networks:
      - hotelbookingnet
    depends_on:
      - eurekaserver
  paymentsvc:
    build: paymentservice
    container_name: paymentsvc
    image: hotelbooking/paymentsvc:1.0.0
    ports:
    - "8083:8083"
    networks:
      - hotelbookingnet
    depends_on:
      - eurekaserver
  notificationsvc:
    build: notificationservice
    container_name: notificationsvc
    image: hotelbooking/notificationsvc:1.0.0
    networks:
      - hotelbookingnet
  eurekaserver:
    build: eurekaserver
    container_name: eurekaserver
    image: hotelbooking/eurekaserver:1.0.0
    ports:
    - "8761:8761"
    networks:
      - hotelbookingnet
    hostname: eureka-service
networks:
  hotelbookingnet:
    driver: bridge
    name: hotelbookingnet
```

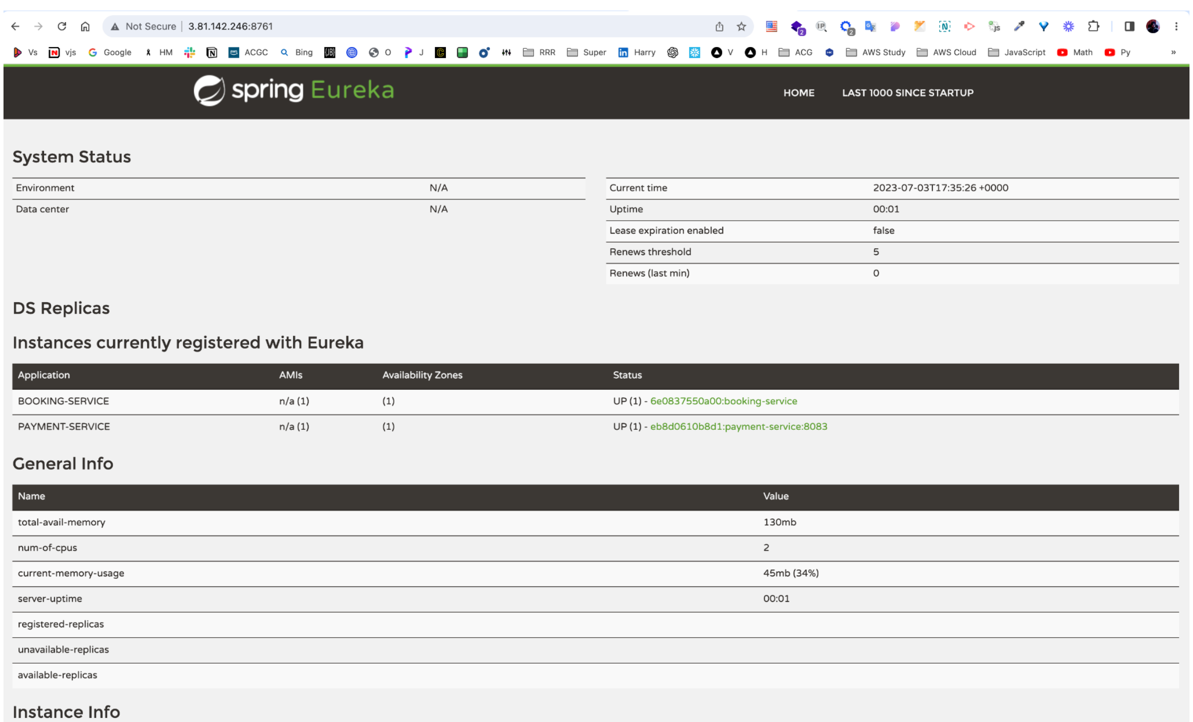
The `docker-compose.yml` file is written in version 3.3 of Docker Compose syntax and defines a multi-container application. It consists of several services, including `bookingsvc` for the booking service, `paymentsvc` for the payment service, `notificationsvc` for the notification service, and `eurekaserver` for the Eureka server (service registry). Each service is configured with properties such as the build context, container name, image, exposed ports, networks, and dependencies. The services are built from their respective Dockerfiles and associated with the `hotelbookingnet` network. Ports are mapped to allow communication between the container and the host machine. The `eurekaserver` service is additionally assigned a hostname so that we don't face Discovery issues. Overall, this `docker-compose.yml` file provides a convenient way to manage and deploy these interconnected services as a single application using Docker Compose.

23. The Project structure would look like this attached image

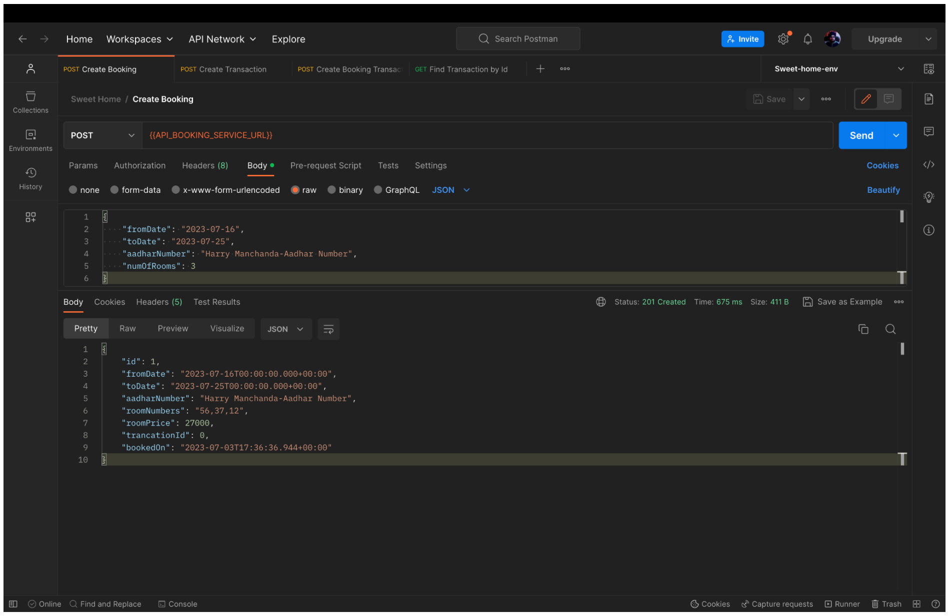


24. Now we come to the one level up directory of the `Sweet-home` folder. In our case, it's `hotel-booking-monorepo` which contains both `Sweet-home` folder and `aws-keys` folder.
25. Now we zip the `Sweet-home` folder which creates `Sweet-home.zip` file, we upload same to our **Main Server** EC2 instance where Docker is installed referring from this [resource]
26. Now we log in to the **Main Server** EC2 instance where Docker is installed referring from this [resource]. After logging in to our EC2 instance, we find and confirm that `Sweet-home.zip` is there respectively.
27. We unzip `Sweet-home.zip` referring from this [resource]. We now have `Sweet-home` folder at our **Main Server** EC2 instance where Docker is installed, now it's about time to not only put the application code in the `/usr` directory but also deploy the project but before we do that let's first start the **Kafka Server** EC2 instance also from another terminal(s).
28. We will open four new terminals for the **Kafka Server** EC2 instance, they will be
    a. First will be to start the **Zookeeper**
    b. Next will be to check whether **Zookeeper** started or not
    c. Now this one will be to start the **Kafka Server**
    d. Next will be to check whether **Kafka Server** started or not

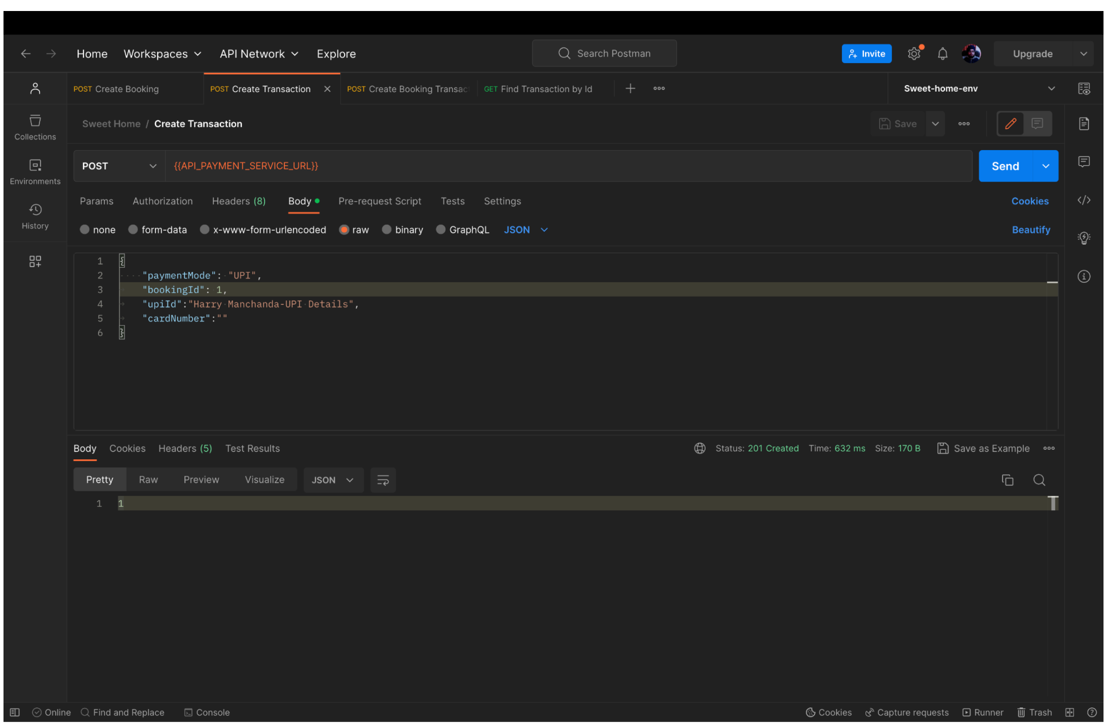
    We referred to this [resource] for setting up the same.
29. Now it's time to go back to the terminal of **Main Server** EC2 instance, we will now run `sudo docker compose up --build` to deploy our Multi container project.
30. Finally our project has been deployed, the Images for all the services are created, and all the containers are running, we can confirm the same by visiting PORT `8761` from our **Main Server** EC2 instance (see attached screenshot).



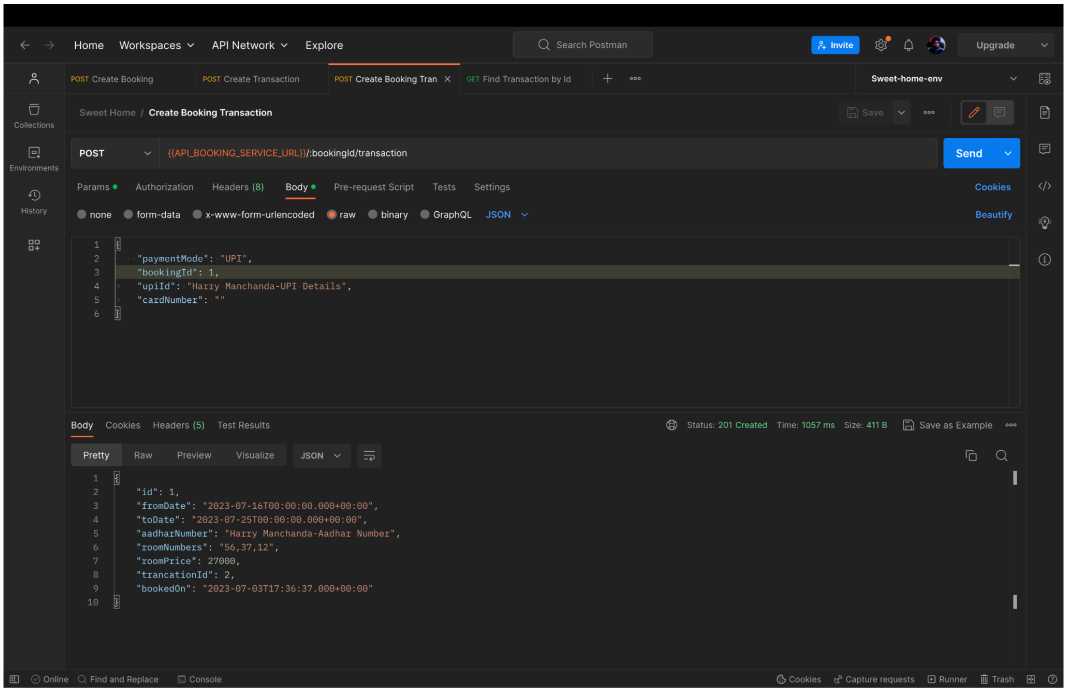
31. Now it’s time to test whether APIs are being hit using postman and returning success response or not.
32. Let’s first test `{MAIN_SERVER_URL}:8080/booking` api endpoint saved as `{{API_BOOKING_SERVICE_URL}}` at postman.



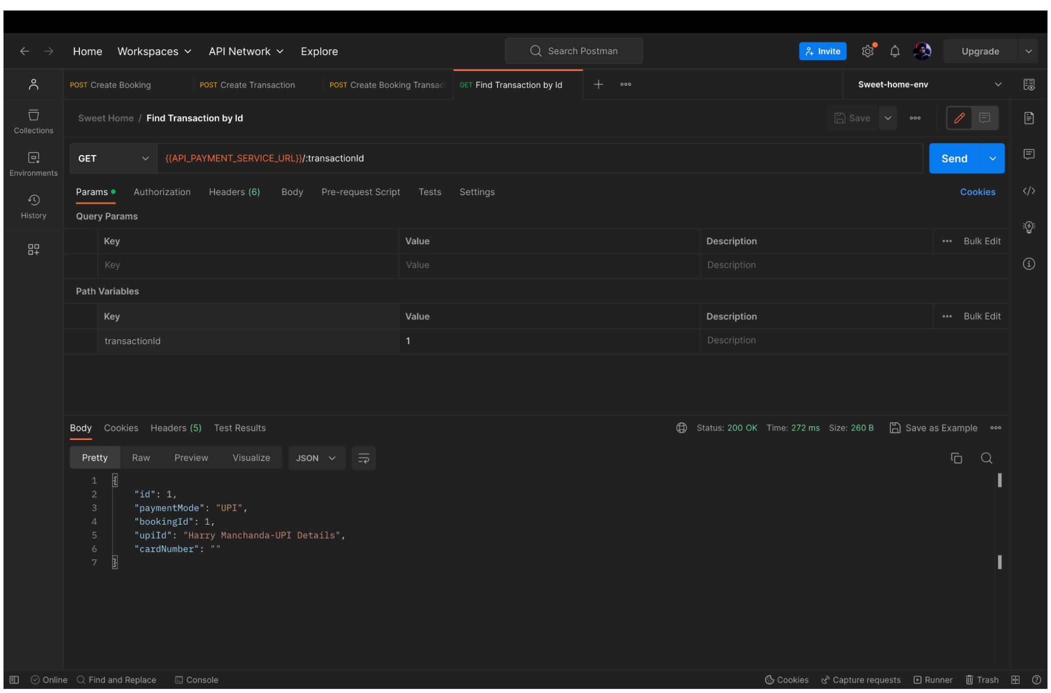
33. Now let’s test `{MAIN_SERVER_URL}:8083/transaction` api endpoint saved as `{{API_PAYMENT_SERVICE_URL}}` at postman.



34. Now let's test `{MAIN_SERVER_URL}:8080/booking/:bookingId/transaction` api endpoint saved as `{{API_BOOKING_SERVICE_URL}}/:bookingId/transaction` at postman.



35. Now let's test `{MAIN_SERVER_URL}:8083/transaction//:transactionId` api endpoint saved as `{{API_PAYMENT_SERVICE_URL}}/:transactionId` at postman.



36. We can confirm that our Project has been deployed successfully!!!

That’s it, if you got here that means the project is running well properly and functionally.